СЕКРЕТЫ ПРОГРАММИРОВАНИЯ ИГР ПЕРЕЗАГРУЗКА

(Andre Lamothe, John Ratcliff,
Mark Seminatore, Denise Tyler)

1994 / 2021

Russian / English

Version 1.2

ВВЕДЕНИЕ

ОТ АВТОРА

В 1996 году, когда с массовыми компьютерами всё только начиналось, мне в руки попала книга Андре Ла Мота «Секреты программирования игр» («Tricks of the Game-Programming Gurus»[1] 1994 Andre Lamothe, John Ratcliff, Mark Seminatore, Denise Tyler). Тогда Андре было 22 года, а мне 20, я был студент и изучал физику и компьютерные науки в одном из университетов СНГ, и плотно «висел» с 14 лет на программировании на всевозможных языках и математике. Книга меня впечатлила, она была понятна. Это книга о складывании высоконагруженных движков, выжимающих всё, что можно из математики, алгоритмов и железа, и я рекомендую её к прочтению и пониманию всем инженерам, причастным к сфере программирования.

Когда спустя почти 20 лет, мне понадобился учебный материал для студентов, я вновь обратился к данной книге. Однако, возникли проблемы – примеры в книге были разработаны под DOS, и не работали под современными операционными системами Windows. Конечно, я не пошёл трудным путём – поставил DOSBox и попробовал компилировать и работать с Borland C++, под которым всё это запускалось и работало 1996-м. Но отладчик и сам IDE в DOS очень неудобны (и как мы это всё делали на нём 1996-м?!), кроме того, DOSBox часто закрывался с ошибками, что в совокупности превращало работу над проектом в ад. Тогда я попробовал поработать с Watcom C/C++ (Open Watcom 1.9) – там тоже всё было грустно – отладчик оказывался работать, вылетая с ошибками.

Тогда я решил портировать примеры из книги под Windows. Порт занял неделю (в свободное от работы время). Все примеры из книги работают, все проверены.

Портирование шло легко, многие программы «заводились» прямо с ходу, в некоторых требовалось внести мелкие правки в код. Это подтверждает, что знания и примеры, изложенные в книге – долговечны, не зависят от программных либо аппаратных средств. Тем и цены. Так же легко примеры книги можно

[1] «Tricks of the Game-Programming Gurus»
https://www.amazon.com/Tricks-Game-Programming-Gurus-Andre-Lamothe/dp/0672305070
ISBN: 978-0672305078, Sams Publishing, 1994

«Секреты программирования игр»
ISBN: 5-88782-037-3, «Питер Пресс», 1995

портировать под Linix либо другие ОС – для этого достаточно переписать несколько функций из DOSEmu.cpp.

Master Mentor, 2019

СЕКРЕТЫ ПРОГРАММИРОВАНИЯ ИГР: ПЕРЕЗАГРУЗКА

КАК ПОЛЬЗОВАТЬСЯ КОДОМ

Чтобы не идти сложным путём, я решил сделать обёртку, подменяющую вызовы функций, не имеющих аналогов в Windows, и отказался от ассемблерных вставок (коих, к слову, было не много).

Получившийся набор инструментов (toolkit), я положит в файлы DOSEmu.cpp и DOSEmu.h

Достаточно скомпилировать эти файлы вместе с исходными кодами программ из книги, и последние – заработают пол Windows. Вот так всё просто.

Важно: при наборе программ обязательно корректно указывайте тип возвращаемого значения функций, иначе работа оптимизированных программ будет некорректной. Если функция не возвращает ничего, укажите тип void.

Среда разработки

В качестве среды разработки исползайте Microsoft Visual Studio 6. Это лучшая версия IDE для обучения программированию. Рекомендую скачать и установить её для работы с toolkit.

Файловая структура

В корне проекта находятся папки:

SOURCES_ORIGINAL – первоначальный код
SOURCES_PORTED – портированный, готовый к компиляции код
SOURCES_EXE – откомпилированные, работающие примеры
SOURCES_DISKS – оригинальные диски, идущие с книгой

Debug – отладочные версии программ (после компиляции)
Release – финальные версии программ (после компиляции)

IDE – CodeBlocks IDE (https://codeblocks.org) вместе с GCC и MinGW для компиляции и выполнения программ.

ENGINES – движки, о которых вы прочтете ниже

TABLES_DOCS_3RD_PARY_SOURCES_ECT – некоторые исходники, которые могут быть полезными при изучении эмуляции DOS

Как запускать примеры в IDE CodeBlocks

1. Разархивировать IDE из
IDE\codeblocks-20.03-mingw-nosetup.zip

2. Запустить CbLauncher.exe IDE и открыть файл:

SOURCES_PORTED\IDE-CodeBlocks-MinGW-Chapters_02_19.workspace

3. Активировать нужный проект, скомпилировать и запустить.

Как запускать примеры в IDE Microsoft Visual C++ 6

1. Запустить IDE, открыть файл:

SOURCES_PORTED\IDE-VisualStudio6-Chapters_02_19.dsw

2. Активировать нужный проект, скомпилировать и запустить.

КАК РАБОТАТЬ НАД КНИГОЙ И С КОДОМ КНИГИ

Как музыкант повышает мастерство игрой на инструменте, так и программист практикуется набором и отладкой кода. Поэтому лучший способ работы с книгой «Секреты программирования игр» – это самостоятельный набор её примеров и проходка по ним отладчиком.

Портированный и исходный код имеет различия в несколько строк на файл кода. Поэтому изучать и разбираться с кодом я рекомендую по книге «Секреты программирования игр», а набирать – пользуясь данным пособием, в котором приведены адаптированные под Windows листинги.

В обязательном порядке возьмите инструмент сравнения файлов и посмотрите на различие в исходном коде и портированном.

КАК ПОРТИРОВАТЬ ПРИМЕРЫ КНИГИ

Файлы:

DOSEmu.h
DOSEmu.cpp

содержат код для эмуляции среды программирования DOS.

При портировании примеров:

1. Создать проект

2. Скопировать файлы .PCX .C и .H файлы из папки с примерами (например, SOURCES_ORIGINAL\CHAP_02), переименовать файлы .C->.CPP и добавить их в проект.

3. Добавить в начало каждого .CPP файла проекта строки:

```
//-------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//-------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//-------------------------------------------
```

Закомментировать строки

```
// #include <graph.h>
// #include <bios.h>
```

Переименовать функцию main() примера в

```
void main2(void)
{
...
}
```

Сравнить содержимое файлов .CPP и .H файлов проекта с их аналогами в SOURCES_PORTED (в нашем примере SOURCES_PORTED\CHAP_02) и внести изменения в .CPP и .H файлы проекта.

Ключевые приёмы работы с toolkit

Изучите содержимое файла DOSEmu.h. В нём находятся с десяток функций и константы с подробным описанием как пользовать их.

DOSEmu toolkit (кратко)

Необходимо представлять работу DOSEmu toolkit, чтобы понимать, как организовать взаимодействие портируемого кода с ним.

DOSEmu toolkit запускает поток, отвечающий за отрисовку графического экрана и ввод данных с клавиатуры, поток, отвечающий за эмуляцию таймера DOS, и передаёт управление функции main2(), которая должна быть определена в портируемом коде.

Память, отображаемая на VGA-мониторе в DOS, начинается с адреса 0xA0000000. В DOSEmu в переменной MEMORY_0xA0000000 хранится указатель на блок памяти, соответствующий графическому экрану эмулятора.

После смены видеорежима следует взять адрес видеобуфера из переменной MEMORY_0xA0000000

```
_setvideomode(_MRES256COLOR);
video_buffer = (unsigned char*)MEMORY_0xA0000000;
```

Чтобы перенести изображение на экран эмулятора нужно скопировать его по адресу указанному в MEMORY_0xA0000000 и вызвать функцию. _redraw_screen(), отвечающую за перерисовку экрана. Поэтому всегда, после копирования данных по указателю MEMORY_0xA0000000, следует вызвать _redraw_screen().

Обновление палитры, работа с клавиатурой, мышью, COM-портами, ведётся через функции _inp(), _outp() – так же, как в DOS.

При обновлении палитры через _outp(0x3c9, ...) сразу происходит перерисовка экрана. Для оптимизации установите режим

```
_set_render_options(TRUE,
RENDER_NOT_REDRAW_IF_BY_PORT_PALETTE_CHANGED)
```

и после обновления всей палитры вызывайте _redraw_screen()

Библиотеку работы со звуковой карты пришлось полностью переписать и упростить. Её функции представляют пустые заглушки, а проигрывание звука ведётся через единственную функцию PlaySound((char*)addr, 0, SND_ASYNC).

Графическое окно DOSEmu ловит нажатие клавиш и перенаправляет их в окно консоли.

В DOS нет функции остановки потока программ, в Windows – это функция Sleep(). Она применяется для организации задержек в циклах.

Работа с портами устройств организовывается просто. «Порт» – это нумерованный контейнер, куда можно записывать байты информации. Подключенному устройству присваивается несколько портов, и следует знать, в каком порядке необходимо записывать (и считывать) информацию в порты, чтобы управлять устройством. Например, чтобы изменить палитру видеокарты, необходимо в порт видеокарты 0x3c6 вывести значение 0xff, затем в порт 0x3c8 номер индекса цвета, который будет обновлён, и затем в порт 0x3c9 последовательно вывести 3 байта, являющихся RGB компонентами цвета в палитре. То есть после вывода в порт 0x3c6 значения 0xff, видеокарта ожидает вывода указанной последовательности байт в определённые порты. На этом построена эмуляция работы устройств DOS.

Настройка отрисовки экрана

Описанные ниже настройки не обязательны (опциональны) и настроены по умолчанию.

Окно вывода графики настраивается через функцию _set_render_options(). Первый её параметр TRUE или FALSE указывает установить либо снять опции, переданные во втором параметре. Второй параметр – опция либо комбинация нескольких опций через битовое ИЛИ | . Например, запретим коррекцию палитры, а так же масштабирование экрана при низких разрешениях:

```
_set_render_options(FALSE,
RENDER_CORRECT_PALETTE | RENDER_SCALE_VGA_SCREEN)
```

Чтобы оптимизировать перерисовку экрана при многократном пользовании графических примитивов из <graph.h> (_moveto(), _lineto(), ...), установите ручной режим перерисовки экрана

```
_set_render_options(TRUE,RENDER_MANUAL_REDRAW)
```

и после отрисовки всех примитивов вызывайте _redraw_screen().

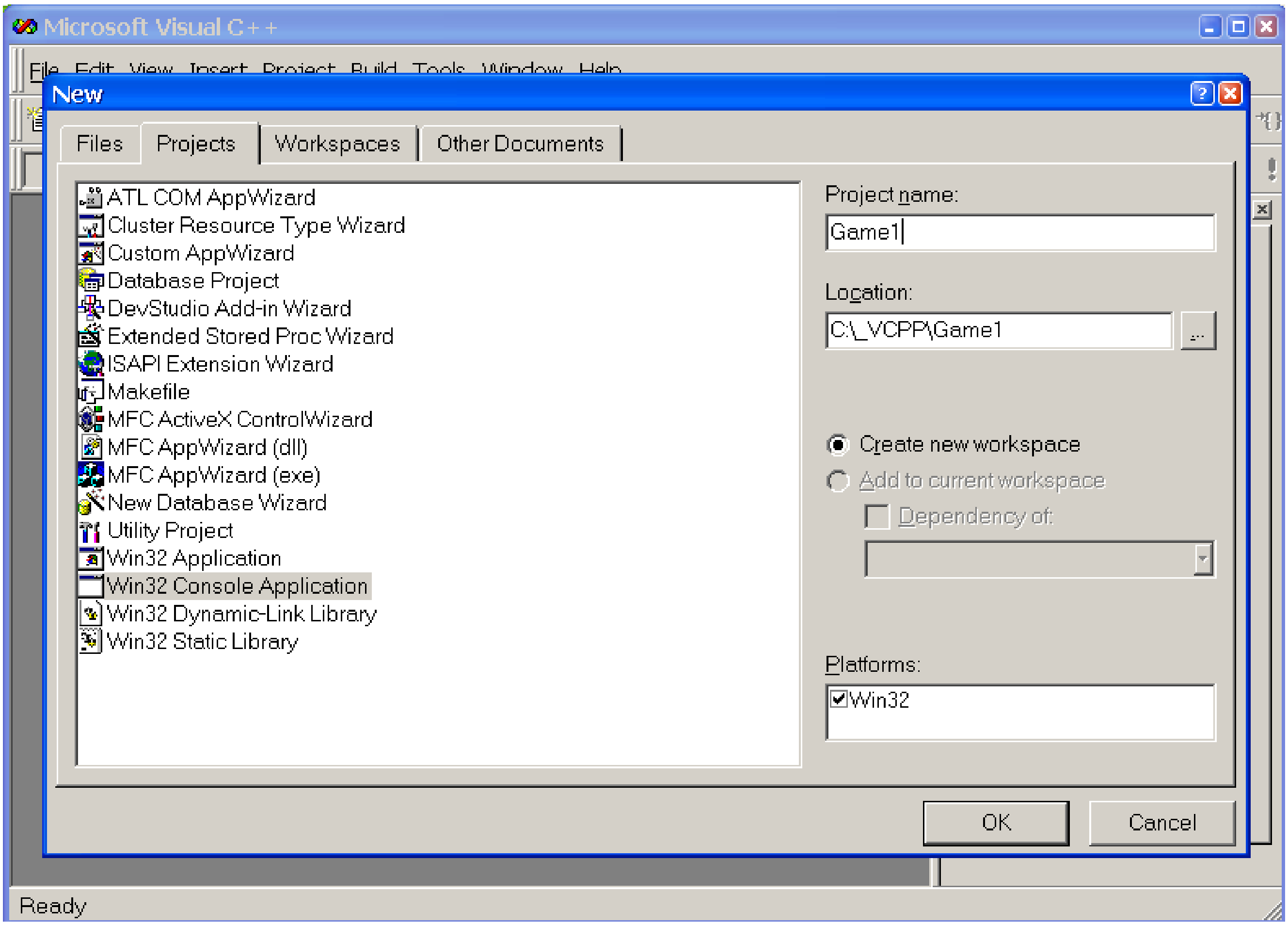
ЛИСТИНГИ ПРОГРАММ

Листинги портированных программ по главам, как они идут в книге «Секреты программирования игр» даны в приложении книги.

САМОСТОЯТЕЛЬНАЯ НАСТРОЙКА IDE VISUAL C++ 6

Чтобы портировать примеры самостоятельно выполните следующие шаги:

1. создайте проект в Visual Studio
File->New->Projects->Win32 console application->An empty project



ATL COM AppWizard
Cluster Resource Type Wizard
Custom AppWizard
Database Project
DevStudio Add-in Wizard
Extended Stored Proc Wizard
ISAPI Extension Wizard
Makefile
MFC ActiveX ControlWizard
MFC AppWizard (dll)
MFC AppWizard (exe)
New Database Wizard
Utility Project
Win32 Application
Win32 Console Application
Win32 Dynamic-Link Library
Win32 Static Library

Project name: Game1

Location: C:_VCPP\Game1

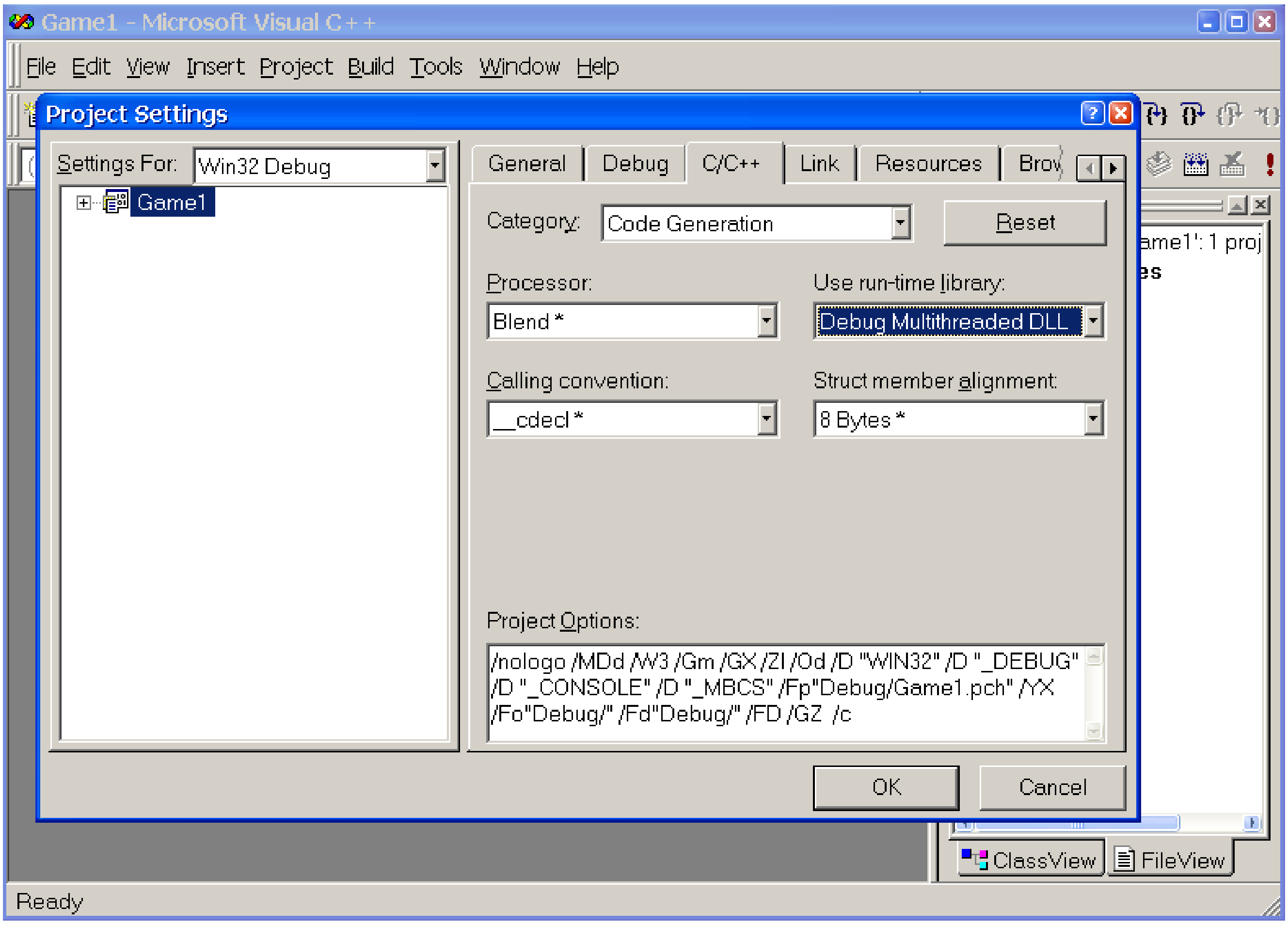
● Create new workspace
○ Add to current workspace
□ Dependency of:

Platforms:
☑ Win32

2. Произведите настройку кодогенерации:

– установите поддержку многопоточности в библиотеках C++

Project->Settings->C++->Category Code Generation->Use run-time library->Debug Multithreaded DLL (либо Debug Multithreaded)


Game1 - Microsoft Visual C++
File Edit View Insert Project Build Tools Window Help
Project Settings
General Debug C/C++ Link Resources Brov
Reset
OK
Cancel
ClassView FileView
Ready

Settings For: Win32 Debug

Game1

Category: Code Generation

Processor: Blend *

Use run-time library: Debug Multithreaded DLL

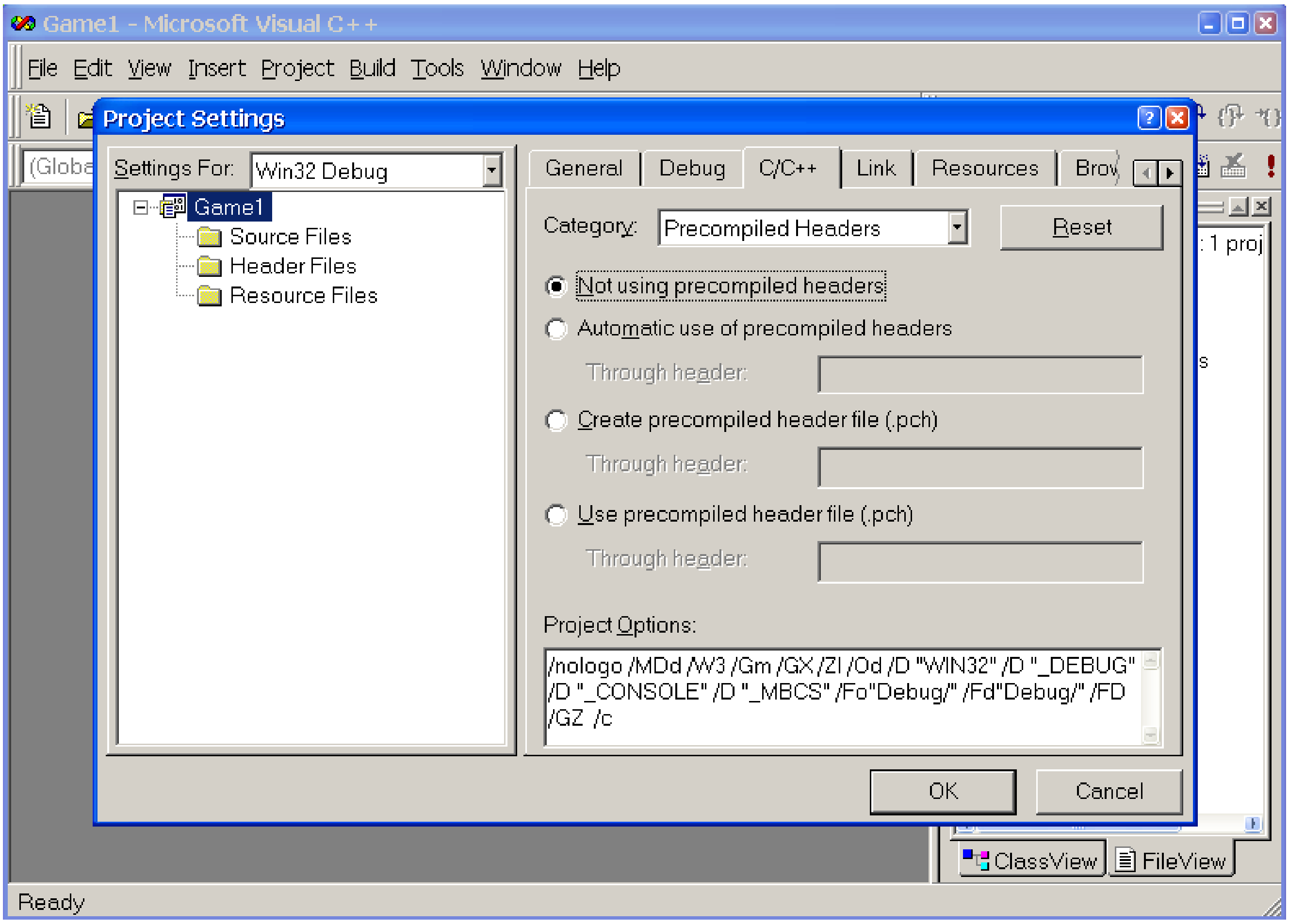
Calling convention: __cdecl *

Struct member alignment: 8 Bytes *

Project Options:

```
/nologo /MDd /W3 /Gm /GX /ZI /Od /D "WIN32" /D "_DEBUG"
/D "_CONSOLE" /D "_MBCS" /Fp"Debug/Game1.pch" /YX
/Fo"Debug/" /Fd"Debug/" /FD /GZ  /c
```

– отключите использование прекомпилируемых заголовков



Game1 - Microsoft Visual C++
File Edit View Insert Project Build Tools Window Help
Project Settings
(Globa
Game1
Source Files
Header Files
Resource Files
General Debug C/C++ Link Resources Brow
Reset
OK Cancel
: 1 proj
s
ClassView FileView
Ready

Settings For: Win32 Debug

Category: Precompiled Headers

● Not using precompiled headers

○ Automatic use of precompiled headers

Through header:

○ Create precompiled header file (.pch)

Through header:

○ Use precompiled header file (.pch)

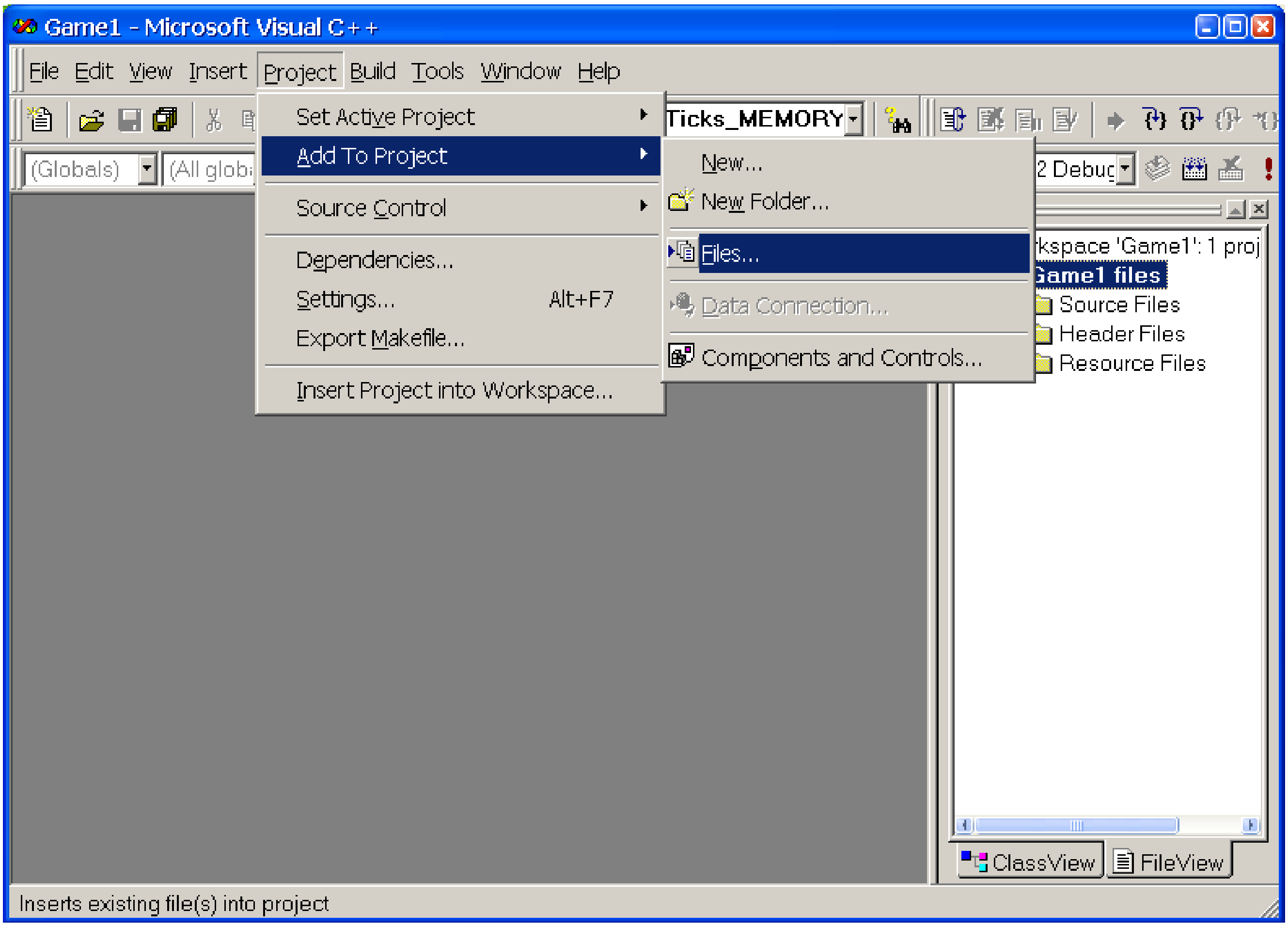
Through header:

Project Options:

```
/nologo /MDd /W3 /Gm /GX /ZI /Od /D "WIN32" /D "_DEBUG"
/D "_CONSOLE" /D "_MBCS" /Fo"Debug/" /Fd"Debug/" /FD
/GZ  /c
```

3. Скопируйте в папку проекта файлы DOSEmu.h DOSEmu.cpp

4. Наберите исходные тексты программ в файлы .CPP и .H, и вместе файлами DOSEmu.h DOSEmu.cpp добавьте их в проект.



Или вы можете скопировать .PCX .C и .H файлы из папки с примерами, переименовать файлы .C->.CPP и вместе с .H файлами добавить их в проект.

5. Скомпилируйте и запустите проект.

Все портированные проекты книги находятся в рабочем пространстве (workspace) Chapters_2_19.dsw

Выполните File-> Open Workspace-> Chapters_2_19.dsw, сделайте активным требуемый проект, откомпилируйте и запустите его.

ДОПОЛНИТЕЛЬНЫЕ МАТЕРИАЛЫ

ЕЩЁ О ДВИЖКАХ (БОНУС)

Движок для игры Another World, Delphine Software, 1991

Книга «Секреты программирования игр» – о движках, а инженер – это раз конструктор движков (центров локализации сложности).

Поэтому в качестве бонуса приложена история создания одного уникального движка под DOS. Уже в конце 1980-х его автор сконструировал виртуальную машину, «крутящую» байт-код, имеющую много поточность, и генерирующую всю векторную графику игровых экранов, одной лишь функцией, рисующей закрашенный многоугольник («полигон»). Игра на этом движке портирована на десяток платформ, запускаемый файл занимает 20 килобайт.

В книги даны две статьи о движке. Одна на русском и на английском. Наполнение их не дублируется, рекомендую к прочтению обе.

В приложении к книге в папке ENGINES находится исходный код данного движка, написанного для игры Another World.

Так же он выложен в репозитории github

https://github.com/fabiensanglard/Another-World-Bytecode-Interpreter

Игра Urban Chaos, Mucky Foot Production, 1999

В 1999 году на платформах Windows, PlayStation, Dreamcast вышла легендарная игра Urban Chaos.

Это игра на все времена, сочетающая выведенные на уровень искусства программный движок, сюжет, художественнее оформление.

Движок таков, что, спустя 20 лет Urban Chaos, не проигрывает современным играм по графике, а современные видеокарты делают графику игры лишь лучше.

В рамках работы над Urban Chaos была написана виртуальная машина, выполняющая скрипты на диалекте языка BASIC.

В приложении к книге в папке ENGINES находится исходный код проекта Urban Chaos.

Так же он выложен в репозитории github

https://github.com/dizzy2003/MuckyFoot-UrbanChaos

АНАЛИЗ ИСХОДНОГО КОДА ANOTHER WORLD

Перевод статьи Fabien Sanglard
http://fabiensanglard.net/anotherWorld_code_review/
По матриалам:
https://habr.com/ru/post/324550/

Движок для игры Another World

Я потратил две недели на реверс-инжиниринг исходного кода Another World (https://github.com/fabiensanglard/Another-World-Bytecode-Interpreter).

Я был потрясён, обнаружив элегантную систему, состоящую из виртуальной машины, интерпретирующей байт-код в реальном времени и генерирующей полноэкранное векторное движение.

Всё это умещалось на гибкий диск ёмкостью 1,44 МБ и работало на 600 КБ ОЗУ.

Исходный код Another World никогда официально не публиковался. Люди, страстно влюблённые в эту игру, выполнили реверс инжиниринг исполняемого файла DOS размером в 20 КБ.

Почему он был таким маленьким? Потому что ANOTHER.EXE – это не игра, а виртуальная машина, выполняющая байт-код и системные вызовы.

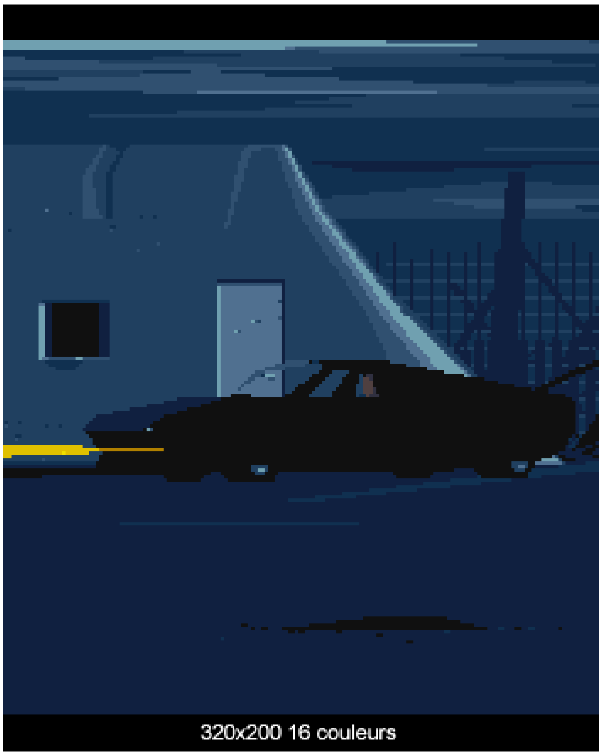
Она выполняет байт-код и использует системные вызовы для выполнения «тяжёлых» задач типа отрисовки, воспроизведения музыки, звуков и управления ресурсами игры.

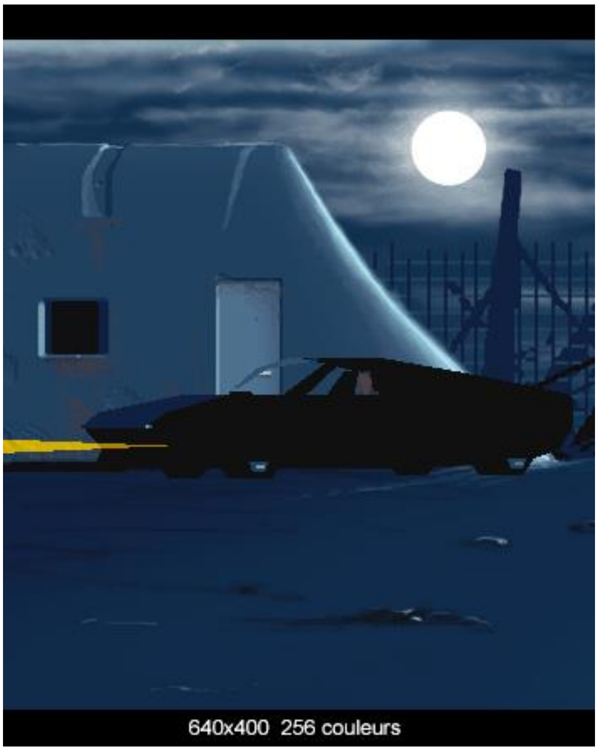
Реализация виртуальной машины под нужную ОС требует меньше усилий, чем перенос программы. Поэтому игра была портирована больше чем на дюжину платформ:

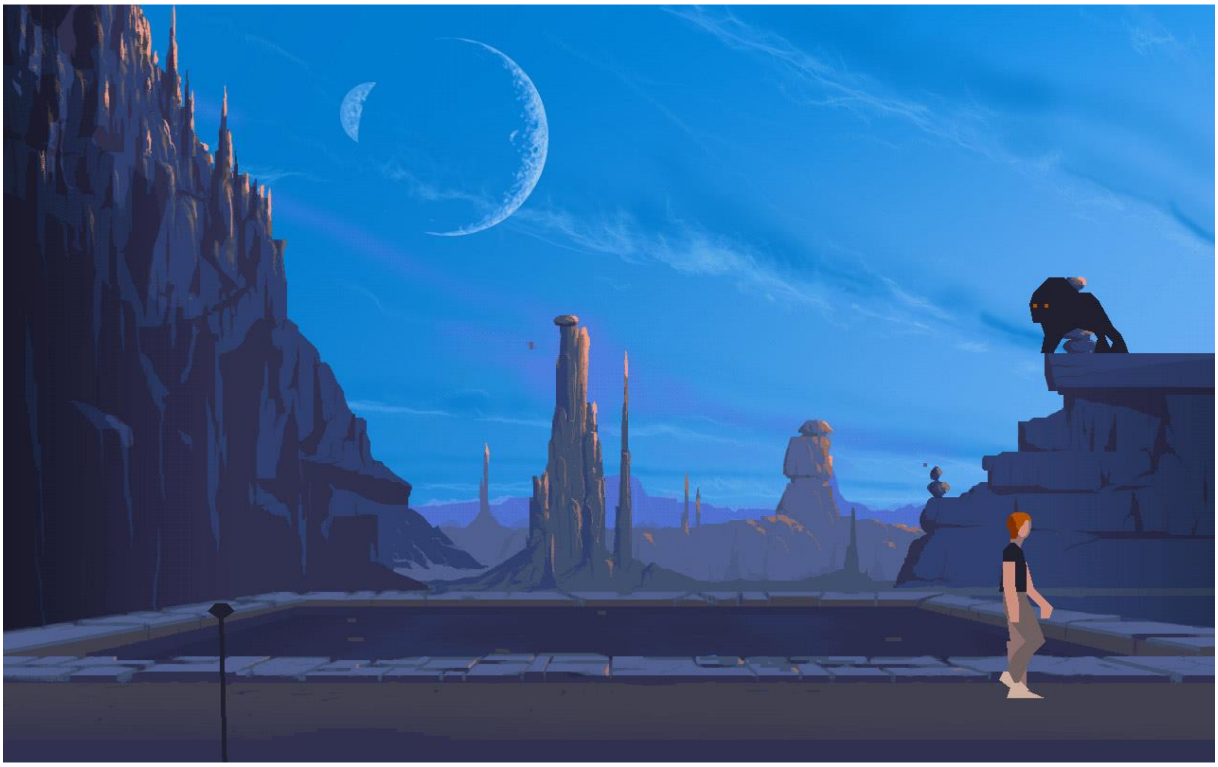
1991 г.: Amiga, Atari ST
1992 г.: Apple IIGS, DOS, SNES, Mega Drive
1993 г.: 3DO
2004 г.: GameBoy Advanced
2005 г.: Windows XP, Symbia OS, Windows Mobile
2011 г.: iOS

Каждый раз нужно было лишь скомпилировать виртуальную машину под ОС – байт-код оставался тем же!

Графика игры при разных разрешениях на разных аппаратных платформах:



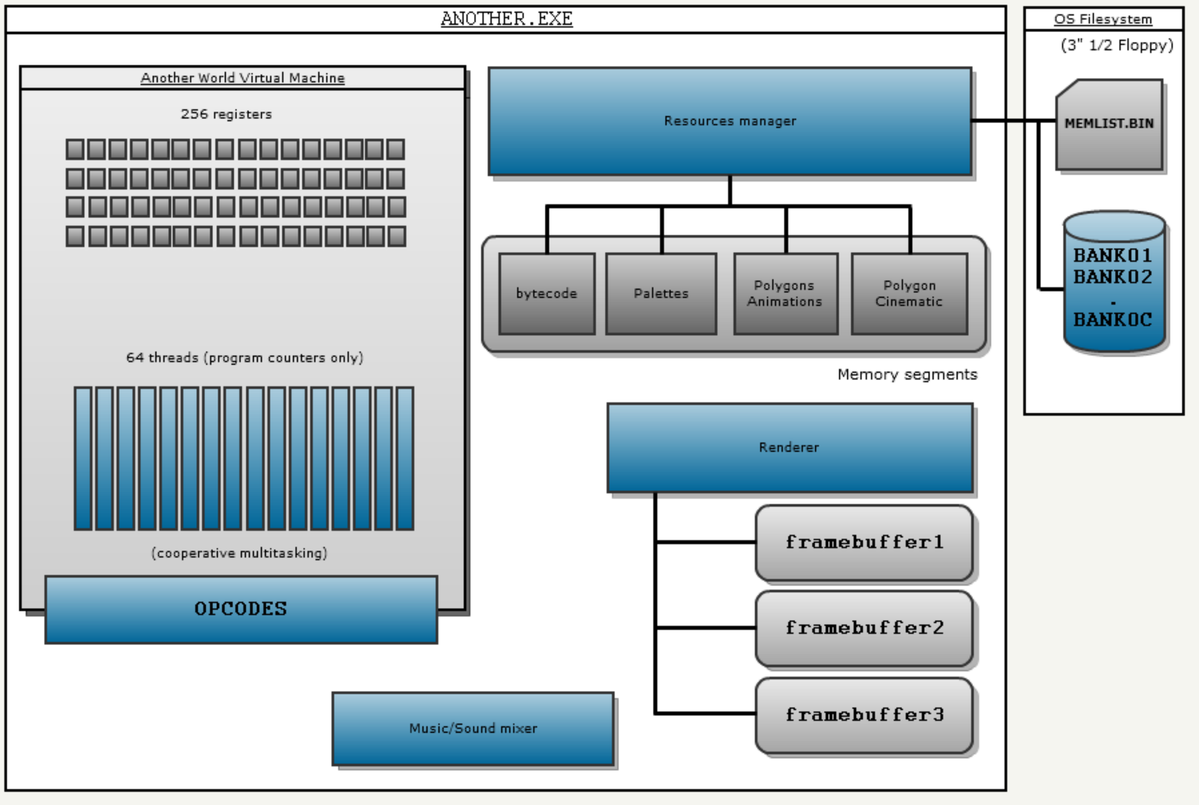




1280x800 resolution

Архитектура

Исполняемый файл имеет четыре модули:



Virtual Machine (виртуальная машина): управляет всей системой.

Resource Manager (менеджер ресурсов): загружает ресурсы с гибкого диска, по запросу виртуальной машины.

Sound/Music mixer (микшер звука/музыки): микширует шумы по запросу ВМ.

Renderer (рендерер): отрисовывает вершины по запросу ВМ

При запуске исполняемый файл устанавливает на начало потока байт-кода виртуальной машины, и та начинает его интерпретирование.

Визуализация

Движок имеет три буфера кадров (framebuffer) – два для двойной буферизации, третий – для ускорения отрисовки (применяется для хранения кадров с неизменной композицией фона и сцены).

Всё отрисовывается с помощью замкнутых закрашенных многоугольников (полигонов), которые отрисовываются по очереди, накладываясь друг на друга.

Количество перерисовок значительное, но сцена генерируется только один раз. Этот фон состоит из 981 полигона:

При двойной буферизация один буфер содержит изображение, выводимое на экран, другой – пользуется для композиции изображения.

Буфер фона генерируется один раз и копируется как подложка для каждого кадра. Если фон изменяется (например, при остановке автомобиля), буфер фона добавляется новыми деталями.

Видео наполнения буферов можно посмотреть здесь:

http://fd.fabiensanglard.net/anotherWorld_code_review/carFull.mov .

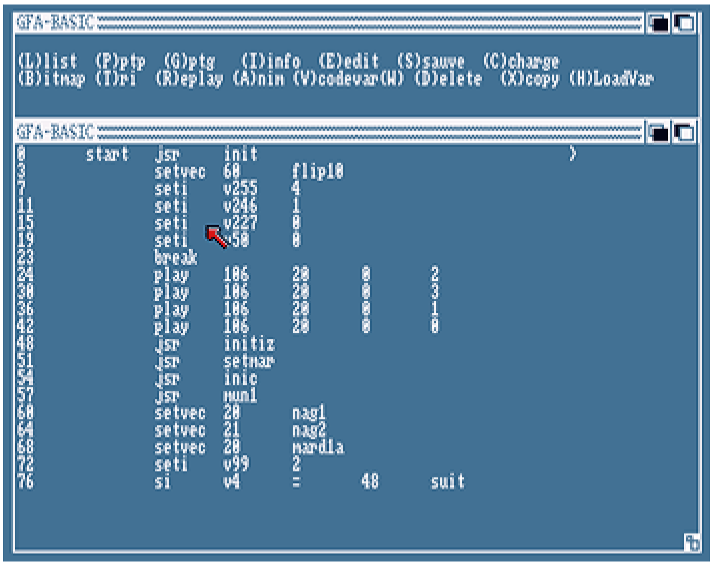
Виртуальная машина (ВМ)

На странице Эрика Шайи подробно объясняется структура ВМ (http://www.anotherworld.fr/anotherworld_uk/another_world.htm) .

В исходниках на github (https://github.com/fabiensanglard/Another-World-Bytecode-Interpreter/blob/master/vm.cpp) можно увидеть во что разворачивается каждый код операции виртуальной машины.

Все байт-коды понятны, за исключением операций визуализации. Хитрость заключается в том, что указатель на источник откуда вершины должны полигона быть прочитаны, встроен в идентификатор кода.

Поток байт-код редактируется при помощи написанного для этого редактора.

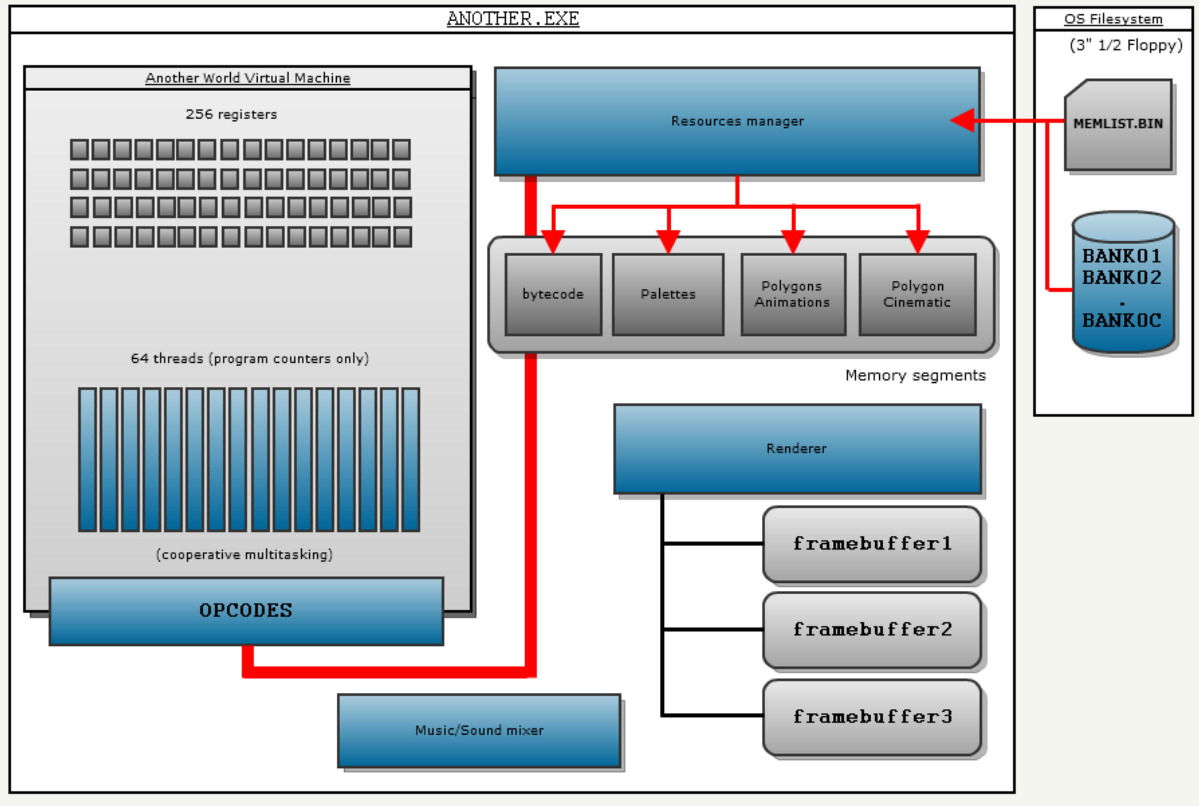


GFA-BASIC
(L)list (P)ptp (G)ptg   (I)info (E)edit (S)sauve (C)charge
(B)itmap (T)ri (R)eplay (A)nim (V)codevar(W) (D)elete (X)copy (H)LoadVar
GFA-BASIC

```
0       start   jsr     init                                    )
3               setvec  60      flip10
7               seti    v255    4
11              seti    v246    1
15              seti    v227    0
19              seti    v50     0
23              break
24              play    106     20      0       2
30              play    106     20      0       3
36              play    106     20      0       1
42              play    106     20      0       0
48              jsr     initiz
51              jsr     setmar
54              jsr     inic
57              jsr     muni
60              setvec  20      nag1
64              setvec  21      nag2
68              setvec  20      mardia
72              seti    v99     2
76              si      v4      =       48      suit
```

Примеры кодов операций ВМ

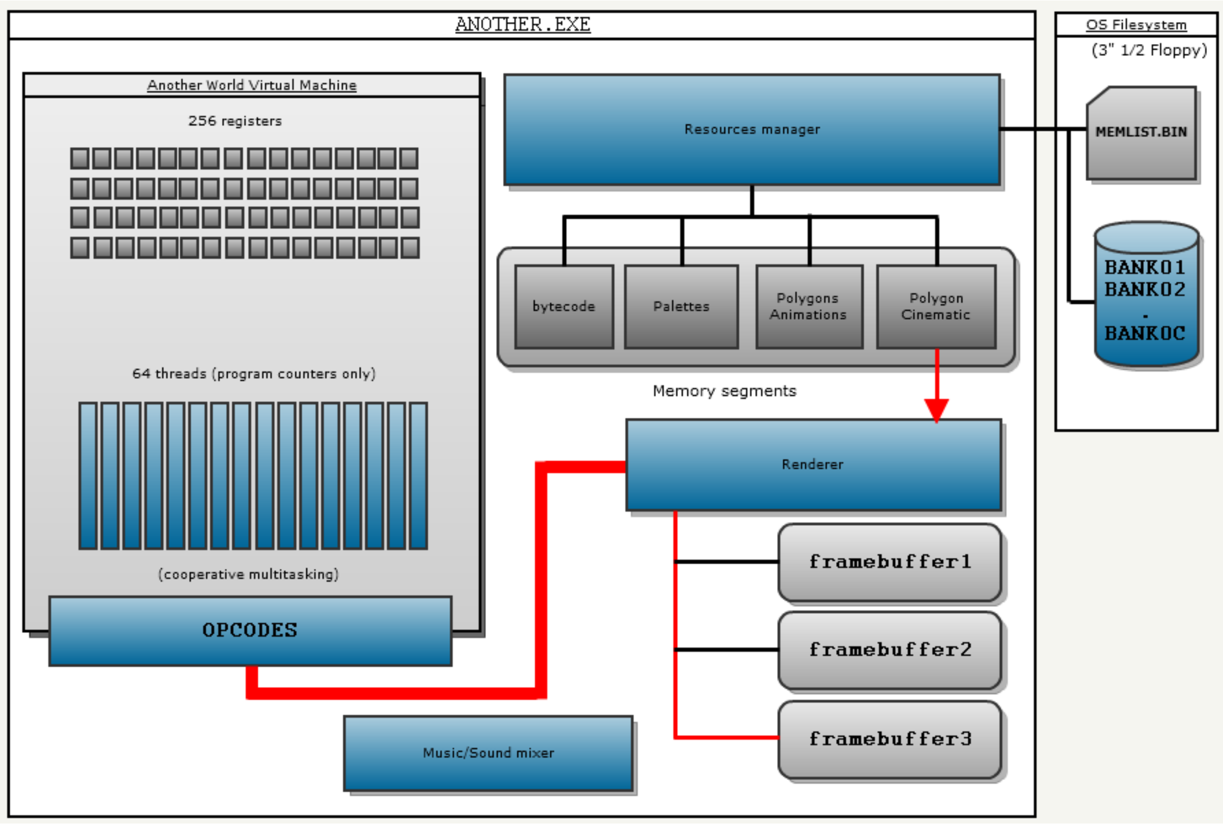
Схема вызова кодов операций виртуальной машины, разворачивающихся в вызовы менеджеру ресурсов для загрузки четырёх сегментов памяти:



Коды операций визуализации немного более сложны, потому что они содержат указания на адреса, из которых нужно считывать вершины.

Выбор источника, откуда рендерер должен считывать вершины (из сегмента полигонов или из сегмента анимаций), кодируется кодом операции (opcodeId).

Ниже – код операции вызывающий рендер, и запрашивающий отрисовку и получение вершин. Выбор целевого буфера кадра – так же код операции:



Управление ресурсами

Ресурсы идентифицируются уникальным целочисленным идентификатором. При запуске менеджер ресурсов открывает MEMLIST.BIN и получает из него набор записей по образцу:

```
typedef struct memEntry_s
{

    int bankId;
    int offset;
    int size;
    int unpackedSize;

} memEntry_t;
```

Если ВМ запрашивает resourceId, то менеджер ресурсов:

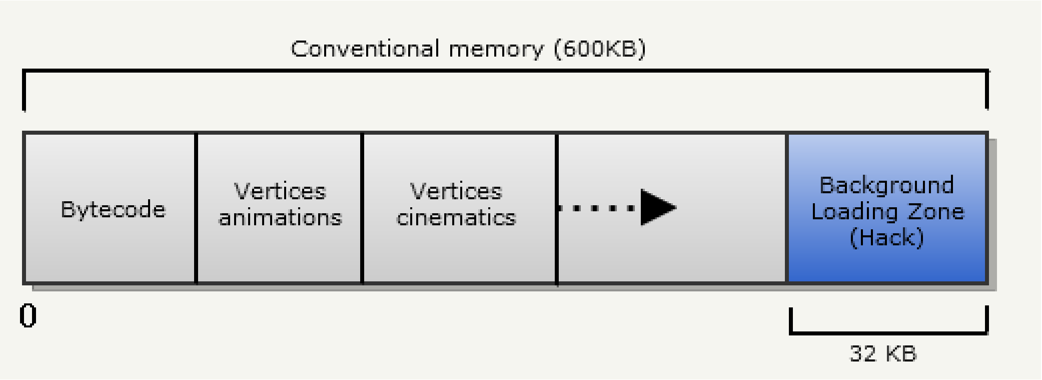
- Находит его запись по идентификатору bankId.
- В хранилище ресурсов, пропускает offset байт и считывает size байт в ОЗУ.
- Если size != unpackedSize, то ресурс распаковывается.

Из 146 ресурсов 120 сжаты: векторная визуализация плюс сжатие (экономия до 62% места) давали огромную выгоду.

Так, начальная заставка длительностью 3 минуты занимает всего 57 510 байт.

Управление памятью

При запуске движок игры получает все 600 КБ памяти DOS. Эти 600 КБ управляются менеджером памяти движка:



Изначально машина хранила байт-код и вершины. Но после двух лет рисования художником игрового мира в редакторе полигонов, игра была далека от завершения. Поэтому Эрик Шайи стал создавать и загружать битовое изображение фона с гибкого диска в специальный буфер (функция void copyToBackgroundBuffer(const uint8 *src)).

Это же применили при выпуске игры под Windows XP. Все фоны были переведены в битовые изображения и загружались напрямую с жёсткого диска без использования рендерера и его полигонов

Исходный код движка

Исходный движка выложен на github.

https://github.com/fabiensanglard/Another-World-Bytecode-Interpreter

Я много работал, чтобы сделать исходный код проще для понимания. Успешного изучения!

ANOTHER WORLD (THE STORY OF CREATION)

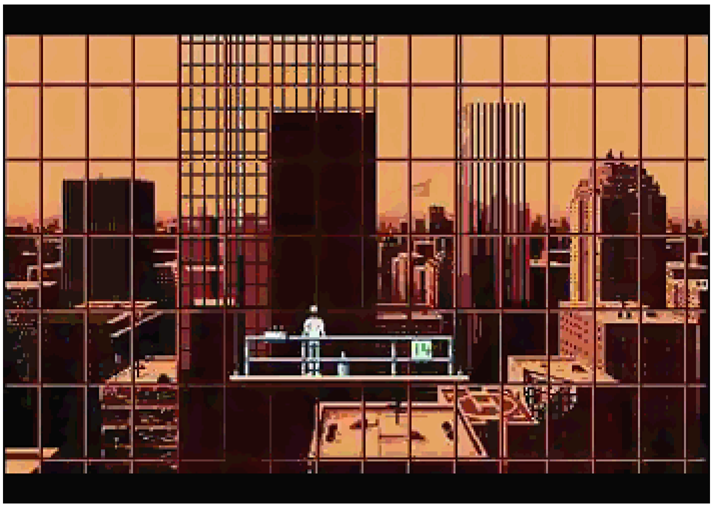
http://www.anotherworld.fr/anotherworld_uk/another_world.htm

Eric Chahi

Genesis

To better understand how the game came into being, I found it useful to retrace my creative progression with a graph. I advise you to take a peek at the time line.

Between 1983 and 1987, I created a series of games, some original, some not, when I was working independently. Then, in 1987-1988, I worked as a graphic designer in the young company Chips. In 1989, I went back to a free-lance position as a graphic designer and animator on "Future Wars", created and programmed by Paul Cuisset. I had stopped programming for about two years, as my last truly original game dated back to 1986, when I started to become lost in ambitious and never finished projects.



Even though I expressed myself freely graphically in "Future Wars", I became frustrated by not being able to create my own games, as I used to. I could have kept on working as a graphic designer on other Delphine Software games. However, in August 1989, when Paul was completing the code for "Future Wars", another game as famous for its spectacular pictures as for its non-interactivity was released: it was the Amiga adaptation of Dragon's Lair. Developers actually managed to store the original videodisk's animations on a floppy: characters filled the visual space, like a cartoon, which was unusual at the time with the reduced sprites' size. The downside of their method was the huge memory storage needed for the game: 6 floppy disks were read during its streaming... When I saw all these animations in flat color, I thought these could be done with vector outlines. That's the sparkle that made me use polygons for 2D animations. This technique has the benefit of using less memory space without any restraints on the animation size. That's the principle used by Flash on the internet.

I knew this principle would be quite perfect for a game with a cinematic atmosphere. The first thing I did was to write a polygon routine on Atari ST in order to make sure this technique would work. I had already worked at the 68000 assembler for a few months, and after a week of writing, performances were getting right, at about 10 displayed polys per 50 frames per second. That was good enough.

Still under the visual influence of Dragon's Lair, I thought I would create a game with very big expressive characters... I thought about many different themes, such as an adventure game in a house haunted by spirits? No, I had already experienced that in "Le Pacte"...

I actually quickly orientated myself towards a theme in which I had worked little but that was always dear to me: Science-fiction. I wanted the player to be immersed in an alien, completely quirky but credible world. Its on this basis I made the introduction, without thinking thoroughly of the development once in the other world, as the separation with the real world would be clear-cut anyway. I kept the game mechanics for later, even though I was already thinking about a 2D game, between "Karateka" and "Impossible Mission" (Epyx, 1984).

The next step was to conceive a creative environment that would exclusively use polygons, and then to realise it in the introduction. Why begin with the introduction, when there is no interactivity? It would have made sense to first work on the game itself, the interactivity being the most sensitive part of a development. My first priority was to achieve what was unknown to me, and I had already created a game with sprites coupled to a scripted mini-language (Infernal Runner). I just wanted to make sure, in the first place, that I could write a polygon editor that would allow me to create complex animations. The introduction is not just a succession of pre-calculated images. Even if its development is predefined, it is sustained by a logic structure where many graphic display scripted layers interact, working together via many tests. Putting the script system to the test allowed me to plan the limits of the future game and thus to make it better.

Improvisation

During my encounter with Costa Gavras for his film called "La Petite Apocalypse", he asked me how I proceeded in creating the game Another World, and if I had already planned the entirety of the game from the beginning. It embarrassed me as it made sense to plan everything in advance and I had worked completely the opposite way. It is clear to me that Another World is the outcome of an educational improvisation !

At the beginning of 1990, the introduction was complete, the first level was being created and I had no clue about the following events, and even less about how the game would end !

On the other hand, I knew precisely what feelings and what look I wanted to communicate throughout the game. That's what ensured the consistency and the direction of the project. I had an emotional guideline and the starting point was well-defined and in tune with what I felt was right. The close elements distinct and the later events vague. I created this game by settling all the details during its creation, like a painter who makes his first sketch and then starts polishing it progressively.

I have to outline that no improvisation has been made however on the game engine and the tools that had been realised in a few months from the beginning of the creation were all made in a stable and almost definitive manner.

It was during this creative process that I fully realized how important rhythm was to storytelling. I unconsciously discovered the duality of interiorization and distancing between the creator and his artwork.

I wanted to communicate a cinematic experience according to two principles:

First, the succession of images, which is the montage, and second, the direction, the dramatic structure.

Contrary to one may believe, I think it is the second point that most characterises Another World. There is a dramatic tension in the game that does not always rely on visual effects, even though the visual effects appear now and then to reinforce efficiently the direction of the game. This game assists the immersion of the player with no exterior elements to the world (score, energy gauge etc.) displayed on the screen.

Realisation

Equipment

An Amiga 500 with 1 MB ram + 20Mb hard drive

A camcorder, a genlock and a video recorder to create the animations according to the rotoscoping technique.

A tape recorder to record sound effects.

"The items used for rotoscoping…"

Software

Commercial applications:

Deluxe Paint, a superb tool for pixel art, used now and then for some backgrounds.

Devpac assembler, used to program the game engine and the polygons outlines.

GFA basic, used to create the game editor specific to Another World.

With the arrival of the 16 bit machines, the C language became more popular in the video game industry, it had many advantages compared to Assembler (portability, structuring and ease of comprehension). Three years before becoming a graphic designer for some time, I tried it on Atari ST with no success. The compiler was shaky, the smallest mistake of programming would result in the machine crashing and the compiling times were exasperating. That experience really put me off. As I always liked simplicity, the evolving languages that didn't need compiling always drew my attention so I intuitively orientated myself towards GFA basic. As simple as a standard Basic and very well-strucured by a proceeding system that manages local variables, a bit like C. GFA is a master work.

Besides, you can extend its features by calling an external program written in Assembler. And that's what I did: I gathered the best of both, first, GFA to quickly program the game editor and second, the Assembler only used for the engine and the graphical features that needed the best performance.

Between the two, there was the Game Data to interpret !

Game editor

I wanted to get a complete tool that would allow me to create and test the whole game without having to compile anything or having to change applications constantly.

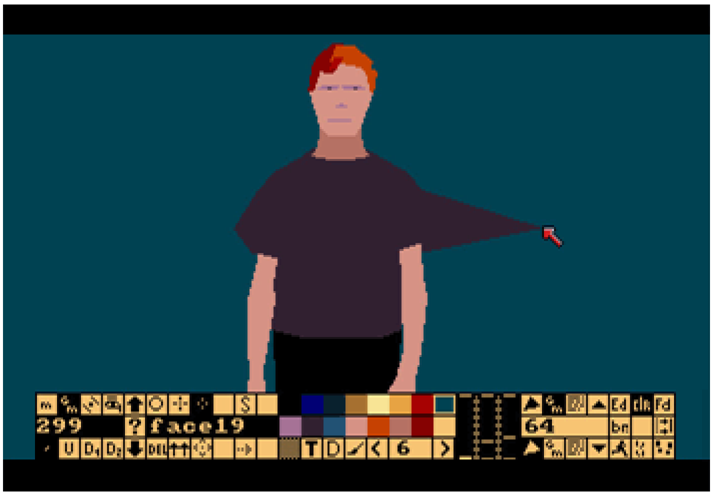
I don't like wasting time with technology and especially with what is specific to an operating system.

So, here is what the first objectives were:

- The game logic should be coded in a language independent from all platforms, without needing compiling or data conversion. So I naturally orientated myself towards the creation of a script interpreter. I developed a mini-language structured in 64 independent execution channels and 256 variables.

- Polygon assembling: each visual display unit of the game was composed by many polygons, so it was essential to gather many polygons in one item, to facilitate their manipulation.

For example; in a character.



- Hierarchical structure of display: The polygon groups could be put together in turn to form another item. This hierarchical structure avoided redundancy through the creation of a priority system. For example, making a specific group for the head of the hero could be re-used in all phases of animation.

Rotoscoping

This technique used in traditional animation consists of filmimg real actors, and then copying each step of their movement on celluloid. It makes very realistic animations. It was used in Another World to create a few animations, especially the ones that were linked to realism, such as the car drifting, that was filmed with a scale model. The shot of the feet in the introduction, and also the character's walking and running.

Trying to do this on a computer at that time was like DIY... the technology was difficult to aquire. The first thought was to use cellophane as carbon copy. It means applying cellophane on a TV screen in order to trace the outlines of a paused video image with a felt pen, and then applying it on the computer screen and reproducing the image with a graphics program. This method isn't too bad for fixed images, like a drawing for example. I had already used it in "Le pacte" for Amstrad, but to actually use this method for a whole video sequence would have been insane.

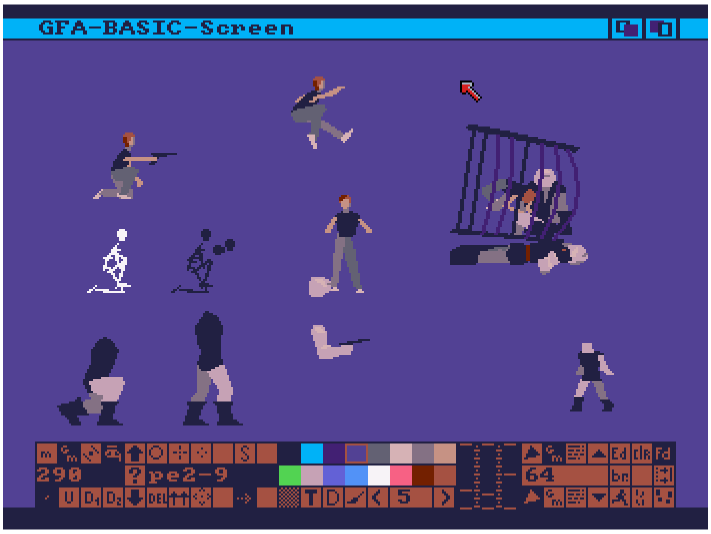
The second idea was to connect a GenLock to an Amiga. It is a device that allows computers to interpret realtime video, coming from a camcorder for example... It sounds simple, but it wasn't the case, because at the time, there was no DVD, no digital camcorder to make a perfect freeze frame. The only device that would cycle through each frame accurately without flickering was a VHS ITT video player that had a digital frame memory. So I had to reproduce all the recorded sequences from the analog camcorder on VHS tape...

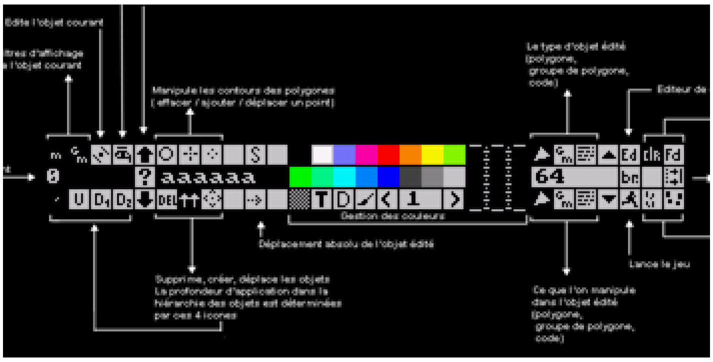
In practice, each video image was copied with polygons manually under the editor. I had to be as quick as possible as the video player would not stay in pause for long, it would stop automatically to keep the tape heads in good condition. It was a time attack race, and in the end, it is especially the introduction that benefited from that technique.

Two video sequences of that time used for the rotoscoping process :
http://www.anotherworld.fr/images/another_world/Rotoscoping_AW_A1.wmv
http://www.anotherworld.fr/images/another_world/Rotoscoping_AW_C2.wmv

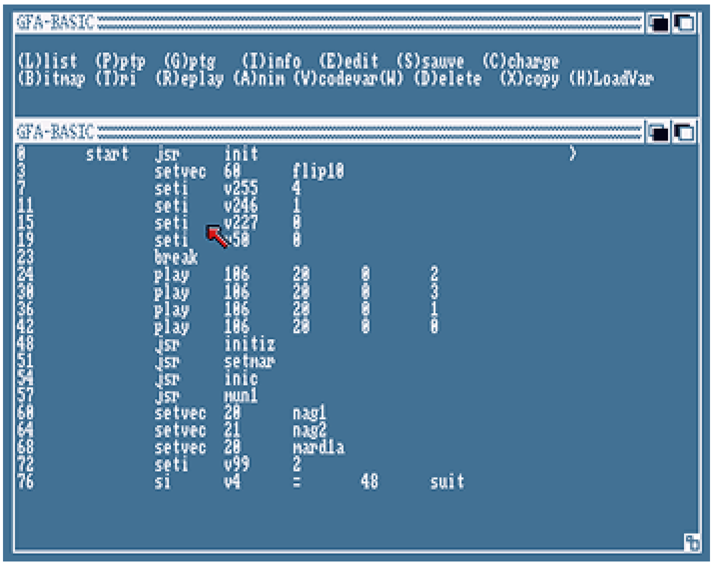
Detailed description of the game engine

I. The polygons editor





II. Script editor:


GFA-BASIC
(L)list (P)ptp (G)ptg (I)info (E)edit (S)sauve (C)charge
(B)itmap (T)ri (R)eplay (A)nim (V)codevar(W) (D)elete (X)copy (H)LoadVar
GFA-BASIC

```
0       start   jsr     init                                  )
3               setvec  60      flip10
7               seti    v255    4
11              seti    v246    1
15              seti    v227    0
19              seti    v50     0
23              break
24              play    106     20      0       2
30              play    106     20      0       3
36              play    106     20      0       1
42              play    106     20      0       0
48              jsr     initiz
51              jsr     setmar
54              jsr     inic
57              jsr     muni
60              setvec  20      nag1
64              setvec  21      nag2
68              setvec  20      mardla
72              seti    v99     2
76              si      v4      =       48      suit
```

This game has been programmed in a made-up language, between Basic and Assembler. At the top window, you'll find the edit functions and at the bottom, the code to edit line by line. The instruction set of the language is very reduced, at around only thirty "words". And within this the game logic has been programmed, including the joystick instructions.

Instruction set

Instructions related to media, graphics, sound, palette:

```
Load "file number"
Loads a file in memory, such as sound, level or image.

Play "file number" note, volume, channel
Plays the sound file on one of the four game audio channels with specific height and volume.

Song "file number" Tempo, Position
Initialises a song.

Fade "palette number"
Changes of colour palette.
```

Clr "Screen number", Color
Deletes a screen with one colour.
Ingame, there are 4 screen buffers.

Copy "Screen number A", "Screen number B"
Copies screen buffer A to screen buffer B.

Show "Screen number" :
Displays the screen buffer specified in the next video frame.

SetWS "Screen number" :
Sets the work screen, which is where the polygons will be drawn by default.

Spr "'object name" , x, y, z
In the work screen, draws the graphics tool at the coordinates x,y and the zoom factor z. A polygon, a group of polygons...

Text "text number", x, y, color
Displays in the work screen the specified text for the coordinates x,y.

Variables and their manipulation

Set.i variable, value
Initialises the variable with an integer value from -32768 to 32767.

Set variable1,variable2
Initialises variable 1 with variable 2.
Variable1 = Variable2

Addi Variable, Value
Variable = Variable + Integer value

Add Variable1, Variable2
Variable1 = Variable 1 + Variable2

Sub Variable1, Variable2
Variable1 = Variable1 - Variable2

Andi Variable, Value
Variable = Variable AND valeur

Ori Variable, Value
Variable = Variable OR valeur

Lsl Variable, N
Makes a N bit rotation to the left on the variable. Zeros on the right.

Lsr Variable, N
Makes a N bit rotation to the right on the variable.

Instruction branch

```
    Jmp Adresse
    Continues the code execution at the indicated address.

    Jsr Adress
    Executes the subroutine located at the indicated address.

    Return
    End of a subroutine.

    Conditional instructions :
    Si (Variable) Condition (Variable ou value) jmp adresse
    Conditional branch,
    If (=Si) the comparison of the variables is right, the
execution continues at the indicated address.

    Dbra Variable, Adress
    Decrements the variable, if the result is different from zero
the execution continues at the indicated address.

    At last but not least, structural instructions :
    Setvec "numéro de canal", adresse
    Initialises a channel with a code address to execute

    Vec début, fin, type
    Deletes, freezes or unfreezes a series of channels.

    Break
    Temporarily stops the executing channel and goes to the next.

    Bigend
    Permanently stops the executing channel and goes to the next.
```

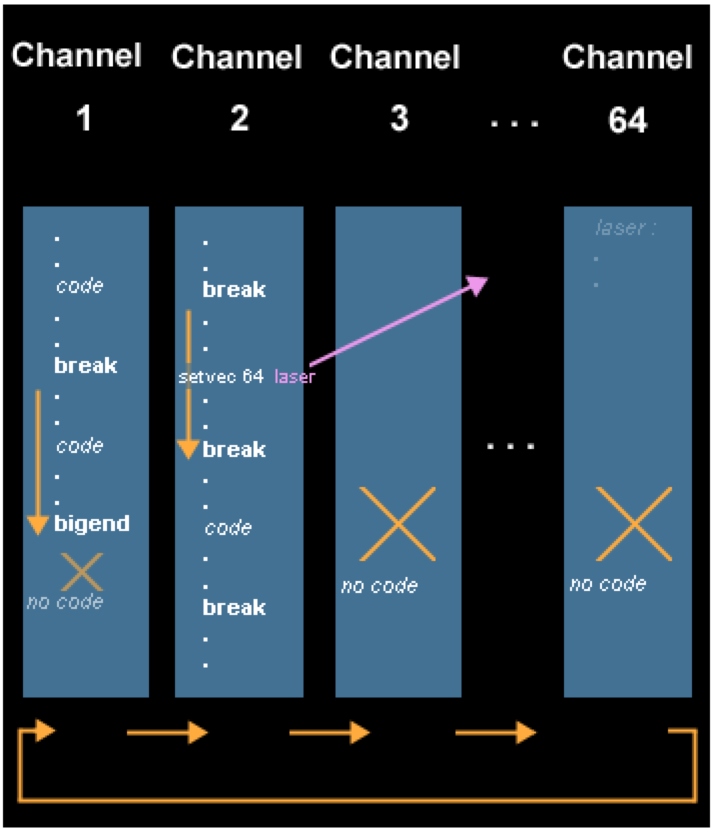
Virtual Machine

The game execution is structured by 64 independent execution channels and 256 variables. A bit like multithread.

Each channel executes a part of the specific code. For example, a channel will manage a character's movements and logic, and the other one its display. Another channel will display birds in the background, and another one will trigger a given animation during a given event.

Channels are executed in order, and the Break instruction indicates to go to the next channel. Once channel 64 is reached, the game frame is displayed and the cycle restarts. Each channel can set up another one with Setvec instruction. In the example below, we can assume that channel 2 manages a character, it decides to shoot and initialises in channel 64 of the loaded code to display and manage a laser shot.

Skim through the image to observe the evolution.



The laser becomes autonomous and disappears when it's out of the screen or hits something, like a door for example, it will then initialise a variable that will make the door explode.

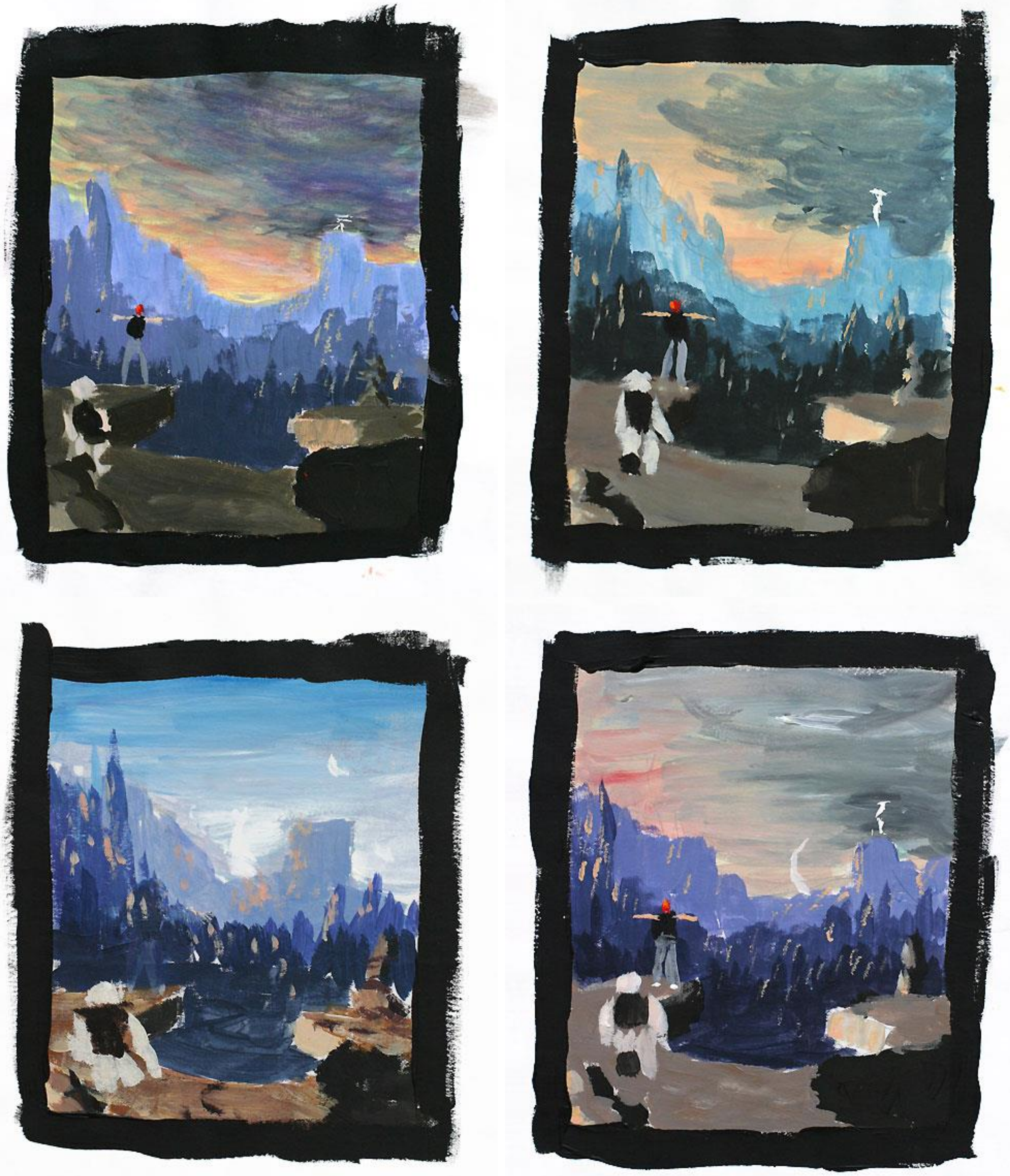
The flow of settings among the channels occurs only through the 254 global variables of the game. Even the joystick positions were written in these variables.

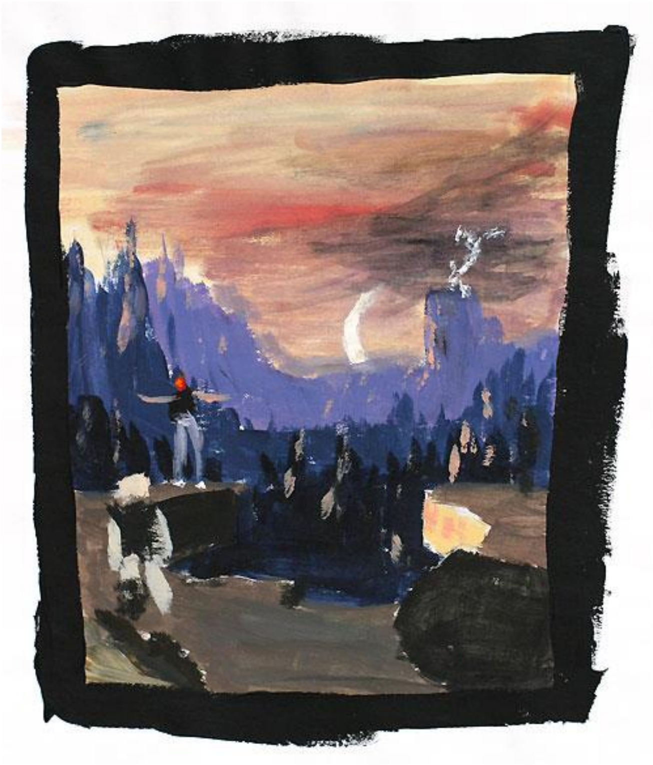
Illustration

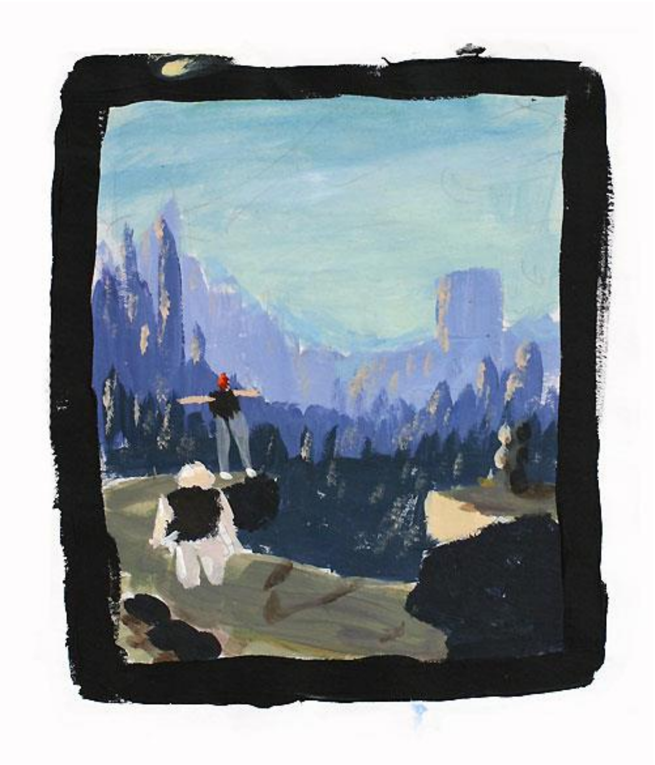
On several occasions I wanted to become an illustrator and make a living from it. Painting and especially fantasy illustration fascinated me. I nearly gave up video games to dedicate myself to it completely, so it was logical that I take care of the game box art. Illustration to me is the extension of a work. It is the first connection the player makes to your game, and so it has to represent it.

You will find below many preparatory sketches.

The composition research was made with Deluxe Paint software. The colour study was painted on paper, without lingering on the accuracy of the drawing, as the impression given by colours was most important.





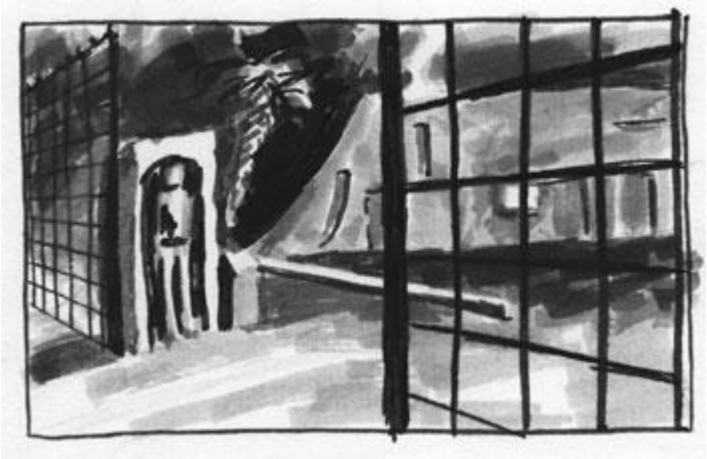


The monochrome bitmap on Amiga was used as a guide, it was printed and then applied on cardboard. Final paint was made in acrylic.

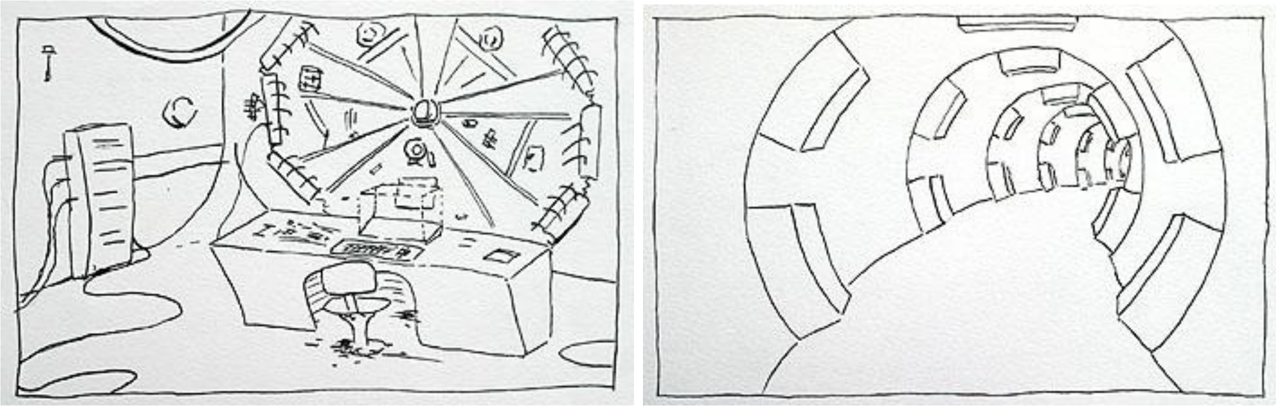
As you can see, the sky color has nothing to do with the blue tones in the game. I hesitated a long time, but I found that, emotionally, the illustration perfectly reflects the feeling of the whole game. Both complete each other through their differences.

Drafts

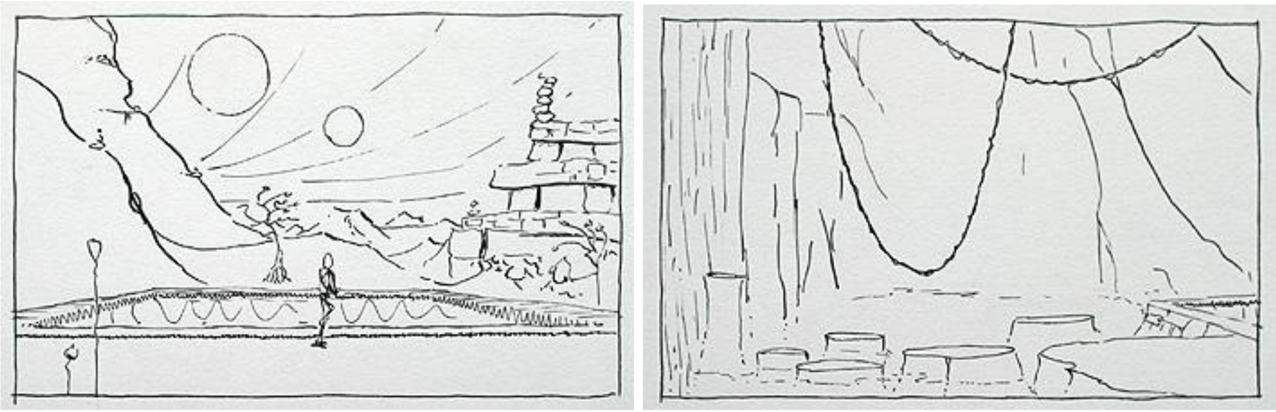
The majority of the game screens were executed without preliminaries directly in the polygon editor. There are, however, some sketches on paper :



The foreground of the introduction. This framing was not satisfactory, it did not emphasise the skid of the car. Also, the presence of the guard at the entry seemed too traditional, without mystery.

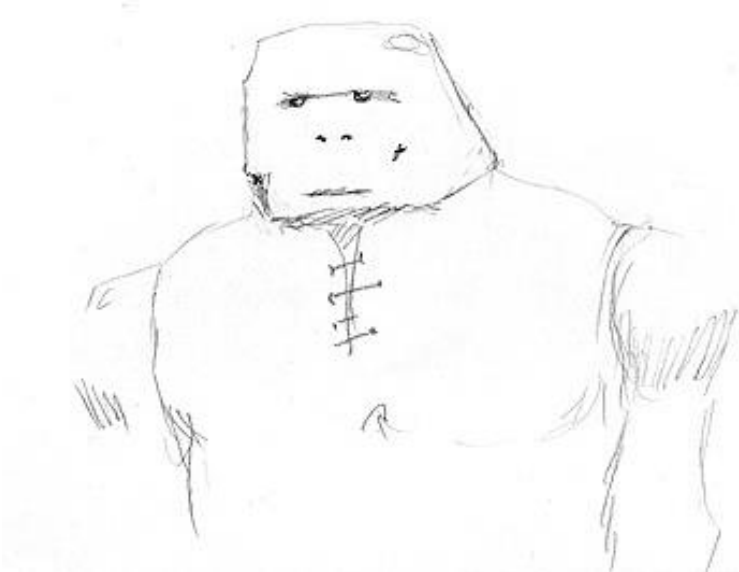


The laboratory… and the cyclotron very close to how they appear today. Essentially, these images are mockups intended to be copied into polygon format.



The second screen of the game was completely restructured.

Also for this screen where Lester swings on the liana vine. As the polygonal graphic style of the game was so unique, I finally realized that it was faster to conceive and develop these images directly in polygons.



This draft dates from the end of the development period.

Aside

At the time of an uncertain period during the creation of Another World, certain doubts pushed me to explore other directions...

This was finally abandoned because it was in dissension with the tonality of the game.

Versions

Initially edited for the first time in November 1991 on Amiga, the game was declined on many media, going through changes, enhancements or deteriorations…

The AMIGA version

The first version, the one with best sound. On the other hand, it has been tested little, which results in a playability that lacks fluidity. That's the drawback of working alone in a garage... Moreover, Delphine Software didn't have testers. As a result, this version is for real hardcore gamers.

This version was also shorter than the others.

The Atari version was identical to the Amiga's, but with less sharp colours and a rougher sound.

The PC DOS version

The articles released at the time criticized the short lifespan of the Amiga version. So Delphine Software suggested I extend it. I had a few ideas left which were enough to make an entire level. However, I didn't want to break the global rhythm of the game, so it was impossible to add anything after the end of the game. The ideal location was just before the arena when the friend rescues Lester at the end of a long dead-end corridor. I decided to reinforce the close relationship between the hero and the alien, by developing their mutual aid. The only problem is that I only had two months to achieve everything. I was back on working 16 hours a day, 7/7. Eventually, that level brought a lot to the game. This version was ported by Daniel Morais.

The console versions SNES and Megadrive

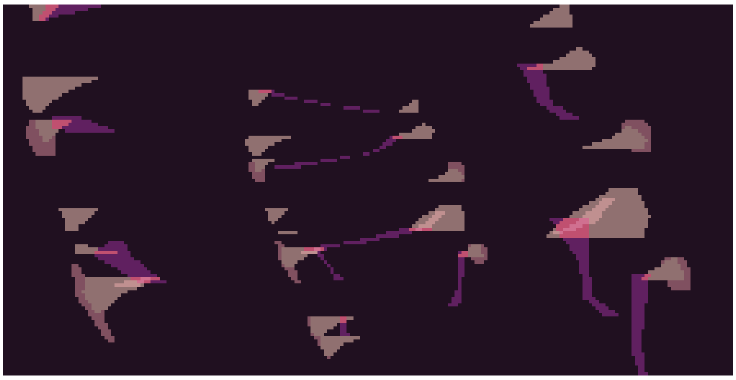
Going through submissions to Nintendo and Sega wasn't an easy task...without speaking about the pressure with Interplay, who was responsible for porting the game engines on those platforms...

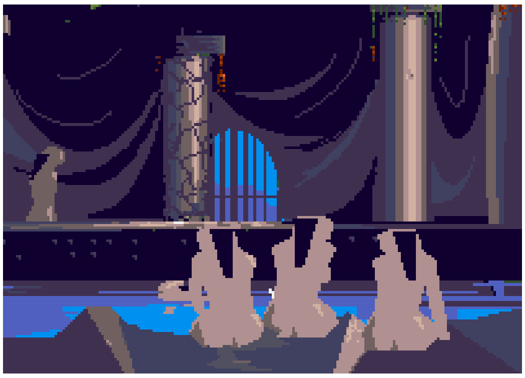
The game was more difficult on consoles than on microcomputer because Interplay really wanted the players to have value for money (a console game is expensive), which implied that the game must have a long lifespan as well. That's why a guard has been added in the prison at the bottom of the lift, and lethal steam jets appeared in the maze-like ventilation system, all of this with a very limited time.

Interplay had imposed on me new songs for all the game levels. They also wanted to replace all the music made by Jean-François. I had yielded for the extra songs, but I wanted to keep the music of the introduction, as it perfecly matched the atmosphere and the animation timing. This became a real struggle, and at the time, we would only communicate by fax, and my letters became firmer each day. Interplay was in a strong position with the development of the game and did not want to back down. So I took drastic measures. I thought of creating an endless fax. A huge fax of a meter long in which I wrote in big letters "keep the original intro music". I would insert it in the fax, enter the number, and when the transmission started, I would tape both ends of the letter together, which would create a circle that went on and on until there was no paper left in the offices of Interplay, at the other end. Even so, all this paper coming out of their machine had little impact. It was Anne-Marie Joassim, Delphine Software's lawyer, who sorted out the situation by applying pressure. She spoke in my favour and demanded once and for all that the original music was kept.

The situation became delicate again when Nintendo of America decided that morally they couldn't release the game due to nudity and presence of blood. Here, there was no other choice but accepting these editorial demands.

I was then forced to withdraw everything that was red and that could eventually look like blood. The smallest reddish bitmap was suppressed or replaced by another colour. Not only the hero's blood, but also any secretion from the bestiary of Another World. The pinkish slobber of the creatures became green during the process.





This scene was too erotic, apparantly. The crack of the naked aliens' bottoms was reduced by 3 pixels...

Mac version

Identical to the PC version, apart from the fact that it supports a higher resolution.

3D0 version

Still developed by Interplay, it benefits from very detailed bitmap backgrounds.

It's not an aesthetic achievement because, as mentioned above, backgrounds are overworked compared to the animations that are made of polygons and thus appear to be flat.

Music had been remade completely, out goes Jean-François and I didn't have my say or the energy to fight, as I was precisely in the "heart of darkness": I still ignored then that this development was to last six years...

megaCD version

It combined two games, on one hand, Another World with CD quality brand new music, made this time by Jean-François, and on the other hand, the sequel named Heart of the Alien.

Interplay insisted in making the sequel in order to make the most of the CD-ROM medium's capabilities. After discussion, I agreed. Rather than making a chronological development related to the first story, I decided that redesigning the game from the alien point of view was excellent, and would make the player discover Another World with other eyes. I could already picture scenes where Lester would be in the background fighting guards, while the player would control the alien in the foreground and then join our first hero, help him, etc... The concept was good but, alas, neither the animations nor the game, entirely developed by Interplay, were up to the job. It was a flop.

GBA version

Unofficially adapted in 2004 by Cyril Corgordan alias Foxy (http://www.foxysofts.com) by creating a reverse engineering of the Atari ST version. I decided in the first place to ban its release in order to make a potential business, but specially authorised its distribution later, in 2005. It required a GBA emulator or a GBA with a Flash cartridge. It was a version for hobbyists. Cyril currently works in the Magic Production company and his C code was the stepping stone for the port of the Symbian mobile version.

At the same time, another GBA port, still unofficial but made this time from the 3D0 version by Gil Megidish, required the original 3D0 CD.

GP32 version

A port from Philippe Simons (http://www.distant-earth.co.uk/awgp32/) , made with more unofficial reverse engineering by another enthusiast, Grégory Montoir. It won a prize during the GBAX 2005 competition.

Mobile and Windows XP versions

Nowadays, I have the privilege to have acquired recently the publishing rights from Delphine. The young company Magic Productions proposed to port the game on mobile phones. I decided to give a little boost to this group of enthusiasts. In collaboration with the crazy programmer who did the reverse engineering of the ST version, Cyril Corgordan, the game engine was coded for mobiles. With hindsight, I found some scenes of the game irritating, so I decided to smooth out the playability by altering the scripts. I even used my Amiga for the purpose. A kind of retro-programming. What a time travel !

To offset the low mobile resolution, I improved the level of shading from the original backgrounds. I really liked the obtained result, so the next natural move was to port the game to Windows. Emmanuel Rivoire was able to increase the resolution to 1280x800 pixels for more detailed images. The definiton was incredible compared to its original format 320x200, which is still available, as the idea was to make a game that was fully customisable, as well as respectful of the past. In one word, it's a collector's edition.

Re-release

Since 2006 Another World's commercial existence has the following supports.

Windows, iOS, Android, Mac, OUYA and also on WII, 3DS, Xbox One, PS3, PS4 / PSVita.

TRICKS OF THE GAME-PROGRAMMING GURU RELOADED

(Andre Lamothe, John Ratcliff,
Mark Seminatore, Denise Tyler)

1994 / 2019

INTRO

FROM THE AUTHOR

In 1996, when the mass computers was just beginning, I got the book Andre La Mothe “Tricks of the game programming gurus“[2] 1994 Andre Lamothe, John Ratcliff, Mark Seminatore, and Denise Tyler). Then Andre was 22 years old, and I was 20, I was a student and studied physics and computer science in the universities in the CIS. I tightly “hung“ with 14 years old in programming in all kinds of languages and mathematics. The book impressed me, it was clear. This is a book about folding high-load engines, squeezing all that is possible from mathematics, algorithms and hardware, and I recommend it to read and understand all the engineers involved in the field of programming.

When after almost 20 years, I needed educational material for students, I again turned to this book. But there are problems: the examples in the book were developed under DOS, and did not work under modern Windows operating systems. Of course, I did not go the hard way: put the DOSBox and tried to compile and work with Borland C++, under which it worked in 1996. But the debugger and the IDE in the DOS are very inconvenient (and how we did it in 1996?!). In addition, the DOSBox was often closed with errors, which together turned the work on the project into hell. Then I tried to work with Watcom C/C++ (Open Watcom 1.9), but the debugger refused to work, closing off with errors.

Then I decided to port examples from the book under Windows. The port took a week (in free time). All examples working all tested.

Porting job was easy, many programs were “started” on the fly, some needed to make minor changes to the code. This confirms that the knowledge and examples set of the book: are durable, do not depend on software or hardware. Therefore, they are valuable. As easy examples of books are porting under Linix or any OS: for this just need to rewrite several functions from the DOSEmu.cpp file.

Master Mentor, 2019

[2] «Tricks of the Game-Programming Gurus»
https://www.amazon.com/Tricks-Game-Programming-Gurus-Andre-Lamothe/dp/0672305070
ISBN: 978-0672305078, Sams Publishing, 1994

«Секреты программирования игр»
ISBN: 5-88782-037-3, «Питер Пресс», 1995

TRICKS OF THE GAME-PROGRAMMING GURUS RELOADED

HOW TO USE THE SOURCE CODE

In order not to go the hard way, I decided to make a wrapper that replaces the calls of functions that have no analogues in Windows, and refused many assembler language inserts (which was a little).

The resulting set of tools (the toolkit) I put in the DOSEmu.cpp and DOSEmu.h files.

It is enough to compile these files together with the source code of the programs from the book, and the programs and the program will run under Windows. That's easy.

Important: when typing programs, be sure to correctly specify the type of the return value of functions, otherwise the operation of optimized programs will be incorrect. If the function returns nothing, specify the void type.

Development environment

Use Microsoft Visual Studio 6 as development environment. This is the best version of the IDE for learning programming. I recommend to download and install it to work with toolkit.

Folder structure

At the root of the project are folders:

SOURCES_ORIGINAL – initial code
SOURCES_PORTED – ported, compile-ready code
SOURCES_EXE – compiled, working examples
SOURCES_DISKS – original disks, going with the book

IDE – CodeBlocks IDE (https://codeblocks.org) with с GCC and MinGW for complie and run programs.

Debug – the debug version of the program (after compilation)
Release – the final version of the program (after compilation)

ENGINES – engines, about which you will read below

TABLES_DOCS_3RD_PARY_SOURCES_ECT – some sources that can be useful in studying the DOS emulation

How to run examples in IDE CodeBlocks

1. Extract IDE from
IDE\codeblocks-20.03-mingw-nosetup.zip

2. Run CbLauncher.exe IDE and open file:

SOURCES_PORTED\IDE-CodeBlocks-MinGW-Chapters_02_19.workspace

3. Activate desired project, compile and run.

How to run examples in IDE Microsoft Visual C++ 6

1. Run IDE and open the file:

SOURCES_PORTED\IDE-VisualStudio6-Chapters_02_19.dsw

2. Activate desired project, compile and run.

HOW TO WORK WITH A BOOK AND BOOK CODE

As a musician increases your mastery by playing in the musical instrument, the programmer is practiced typing and debugging code. Therefore, the best way to work with the book “Tricks of the Game-Programming Gurus”: type the examples yourself and go through all the procedures under the debugger.

Ported and the source code has differences in a few lines of the code. But to study and understand the algorithms, I recommend read algorithms in the book "Secrets of programming games", and type using this guide, which are listings adapted for Windows.

Also you can get comparison tool and look at the difference in the source code and ported (mirrored) files.

HOW TO PORT THE EXAMPLES OF BOOK

Files:

DOSEmu.h
DOSEmu.cpp

implement DOS programming environment.

1. Create a project

2. Copy files .PCX .C and .H files from the examples folder (e.g. SOURCES_ORIGINAL\CHAP_02). Then rename the files .C->.CPP and add them to the project.

3. Add to the beginning of each .CPP project file strings:

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------
```

Comment strings:

```
// #include <graph.h>
// #include <bios.h>
```

Rename the main() function of the example to

```
void main2(void)
{
...
}
```

Compare the contents .CPP .H project's files with their analogues in SOURCES_PORTED (in our example SOURCES_PORTED\CHAP_02) and make changes into .CPP .H files of the project.

Key techniques for working with toolkit

Study the contents of the DOSEmu.h file. It contains a dozen functions and constants with a detailed description how to use them.

The DOSEmu toolkit (briefly)

You must know how the DOSEmu toolkit a working, for understand how to organize a integrating code and toolkit.

The DOSEmu toolkit starts the thread that drawing the graphical screen and catch keyboard input, the thread that emulate the DOS timer. Then toolkit passes control to the main2() function, that must be defined in the ported code.

Memory, displayed on a DOS VGA monitor, starts at address 0xA0000000. In toolkit a variable MEMORY_0xA0000000 store a pointer to a memory block that contains the graphical screen of the emulator. After changing the video mode, you should get the address of the video buffer from the MEMORY_0xA0000000 variable.

```
_setvideomode(_MRES256COLOR);
video_buffer = (unsigned char*)MEMORY_0xA0000000;
```

For clone the image to the emulator screen, just copy image bytes to the address specified in MEMORY_0xA0000000 and call the function _redraw_screen(). This function do redrawing the screen. You can always call _redraw_screen() after copying the image bytes to the MEMORY_0xA0000000 pointer.

Updating the palette, working with the keyboard, mouse, COM-ports, is carried out through the functions _inp(), _outp(), as well as in the DOS.

When updating the palette via _outp(0x3c9, ...) redrawing of the screen doing immediately. To optimize, set the mode

```
_set_render_options(TRUE,
RENDER_NOT_REDRAW_IF_BY_PORT_PALETTE_CHANGED)
```

and after updating the palette call _redraw_screen().

The library for work with the sound card had to be completely rewritten and simplified. Its functions represent empty stubs, and the sound is played through a single function PlaySound((char*)addr, 0, SND_ASYNC).

The DOSEmu graphical window catches keypresses and redirects them to the console window.

In epy DOS there is no function to delay the program flow, in Windows it is a function Sleep(). It is used to organize delays in programmes cycles.

Work with hardware ports is organized simply. A “port“ is a numbered container where bytes of information may be written. The connected device is assigned multiple ports, and you should know the order to write and read information to the ports to control the device. For example, to change the palette of the video card, you need to put to port graphics card 0x3c6 the value of 0xff, then to port 0x3c8 the index (a number) of the color that will be updated, and then to port 0x3c9 sequentially output the 3 bytes which are the RGB color in the palit. That is, after the 0xff value is output to port 0x3c6, the graphics card waits for the specified byte sequence to specified ports be output. This is the basis for emulation of DOS devices.

Setting up screen rendering

The settings described below are optional and are set by default correctly.

Window graphics output is configured through the function _set_render_options(). Its first parameter TRUE or FALSE specifies to set or remove the options passed in the second parameter. The second parameter is an option or a combination of several options via bitwise operator OR | . For example, you can prevent palette correction and screen scaling by call

```
_set_render_options(FALSE,
RENDER_CORRECT_PALETTE | RENDER_SCALE_VGA_SCREEN)
```

To optimize the screen redrawing when using graphics primitives from <graph.h> (_moveto(), _lineto(), ...), first set the manual redrawing mode

```
_set_render_options(TRUE,RENDER_MANUAL_REDRAW)
```

then draw all primitives and call _redraw_screen().

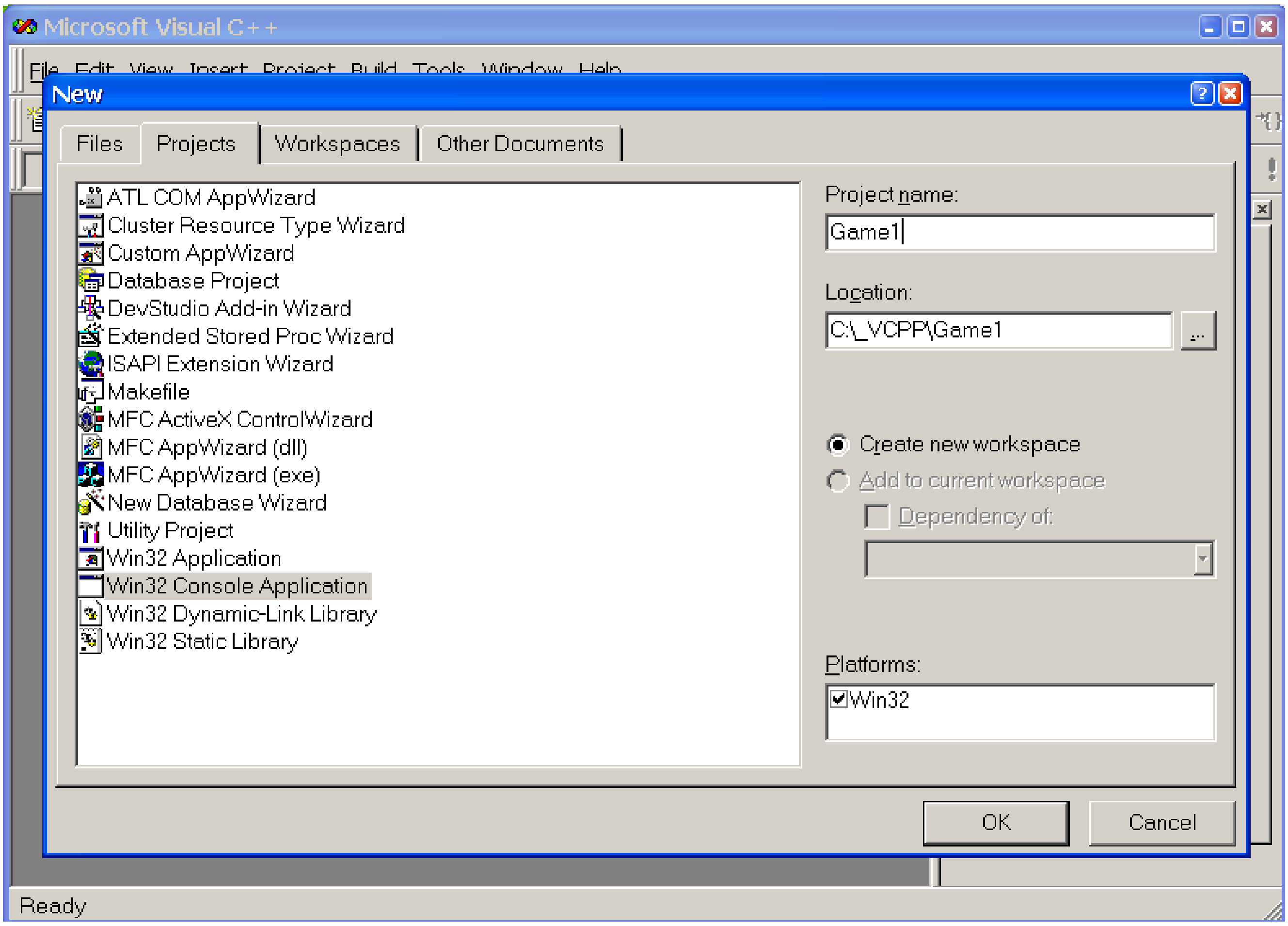
PROGRAMS LISTINGS

Listings of ported programs as they go in the "Tricks of the Game-Programming Gurus" book are given in the Appendix (chapter by chapter).

PREPARING IDE VISUAL C++ 6 YOURSELF

To port the examples yourself, follow these steps:

1. Create a project in Visual Studio
File->New->Projects->Win32 console application->An empty project



ATL COM AppWizard
Cluster Resource Type Wizard
Custom AppWizard
Database Project
DevStudio Add-in Wizard
Extended Stored Proc Wizard
ISAPI Extension Wizard
Makefile
MFC ActiveX ControlWizard
MFC AppWizard (dll)
MFC AppWizard (exe)
New Database Wizard
Utility Project
Win32 Application
Win32 Console Application
Win32 Dynamic-Link Library
Win32 Static Library

Project name: Game1

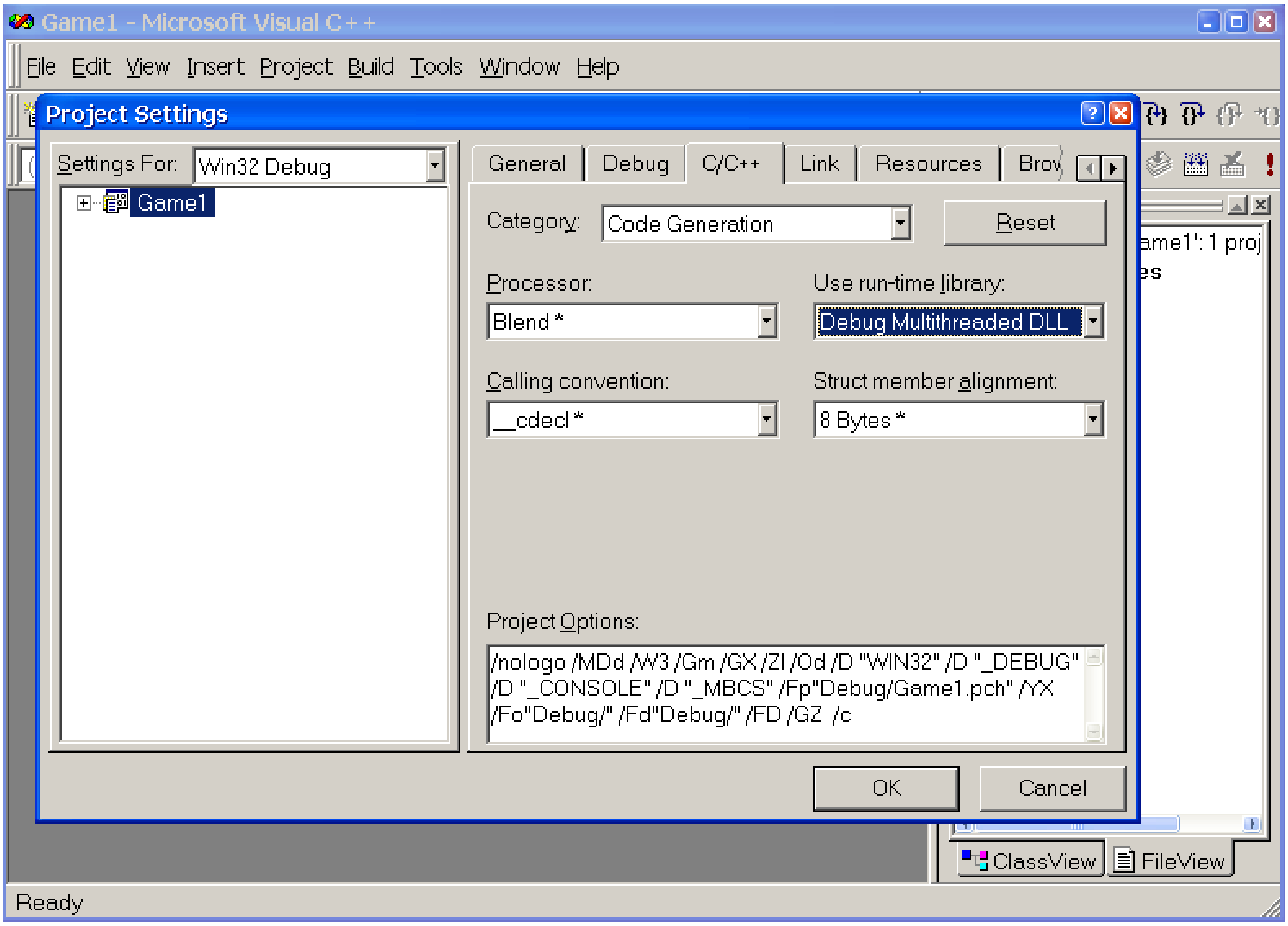
Location: C:_VCPP\Game1

● Create new workspace
○ Add to current workspace
□ Dependency of:

Platforms:
☑ Win32

2. Adjust the code generation:

– set multithreading support in C++ libraries
Project->Settings->C++->Category Code Generation->Use run-time library->Debug Multithreaded DLL (or just Debug Multithreaded)



Settings For: Win32 Debug

Game1

Category: Code Generation

Processor: Blend *

Use run-time library: Debug Multithreaded DLL

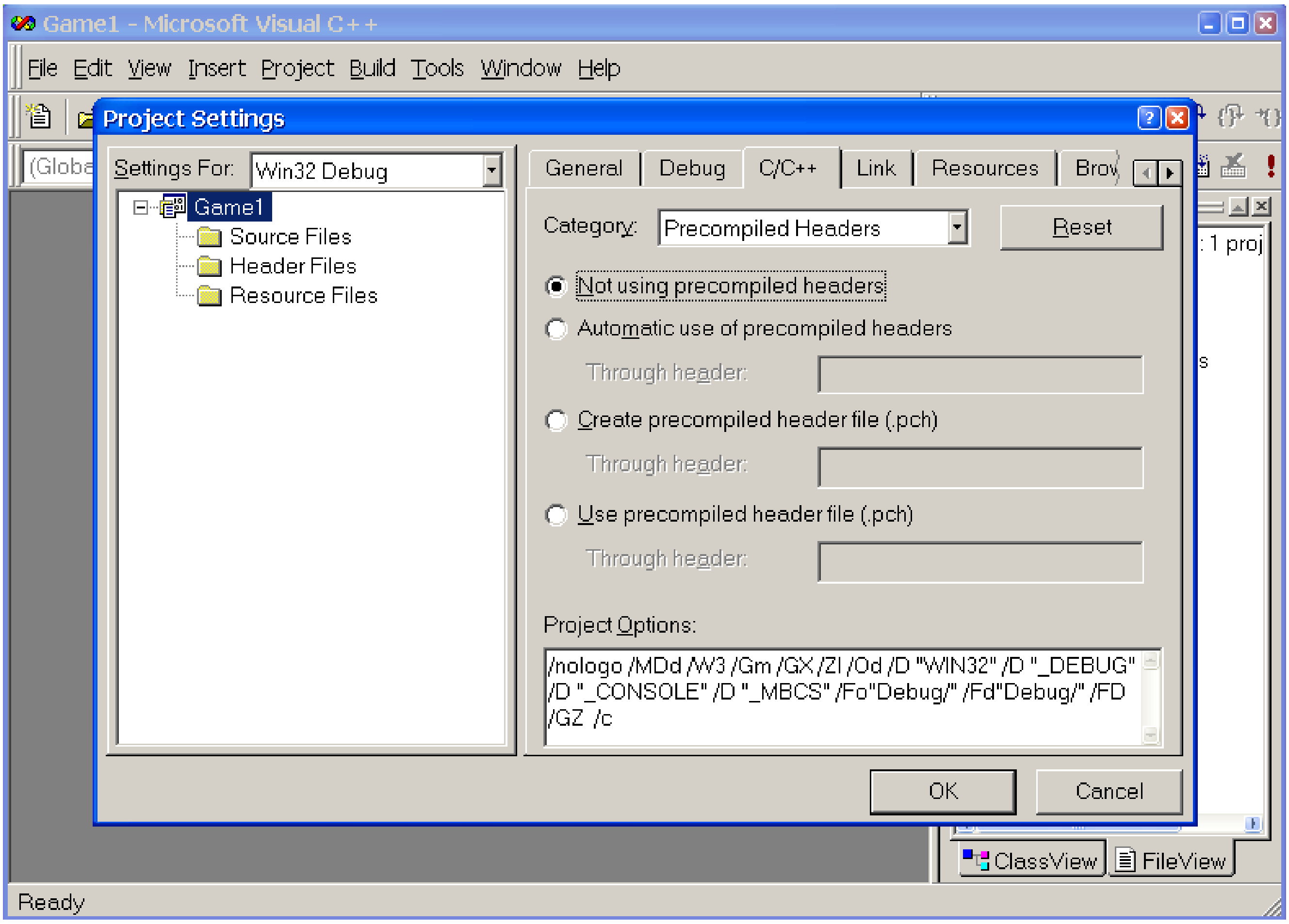
Calling convention: __cdecl *

Struct member alignment: 8 Bytes *

Project Options:

```
/nologo /MDd /W3 /Gm /GX /ZI /Od /D "WIN32" /D "_DEBUG"
/D "_CONSOLE" /D "_MBCS" /Fp"Debug/Game1.pch" /YX
/Fo"Debug/" /Fd"Debug/" /FD /GZ  /c
```

– disable the use of precompiled headers



Game1 - Microsoft Visual C++
File Edit View Insert Project Build Tools Window Help
Project Settings
Settings For: Win32 Debug
Game1
Source Files
Header Files
Resource Files
General Debug C/C++ Link Resources Brow
Reset
Through header:
Through header:
Through header:
OK
Cancel
ClassView
FileView
Ready

Category: Precompiled Headers

● Not using precompiled headers

○ Automatic use of precompiled headers

○ Create precompiled header file (.pch)

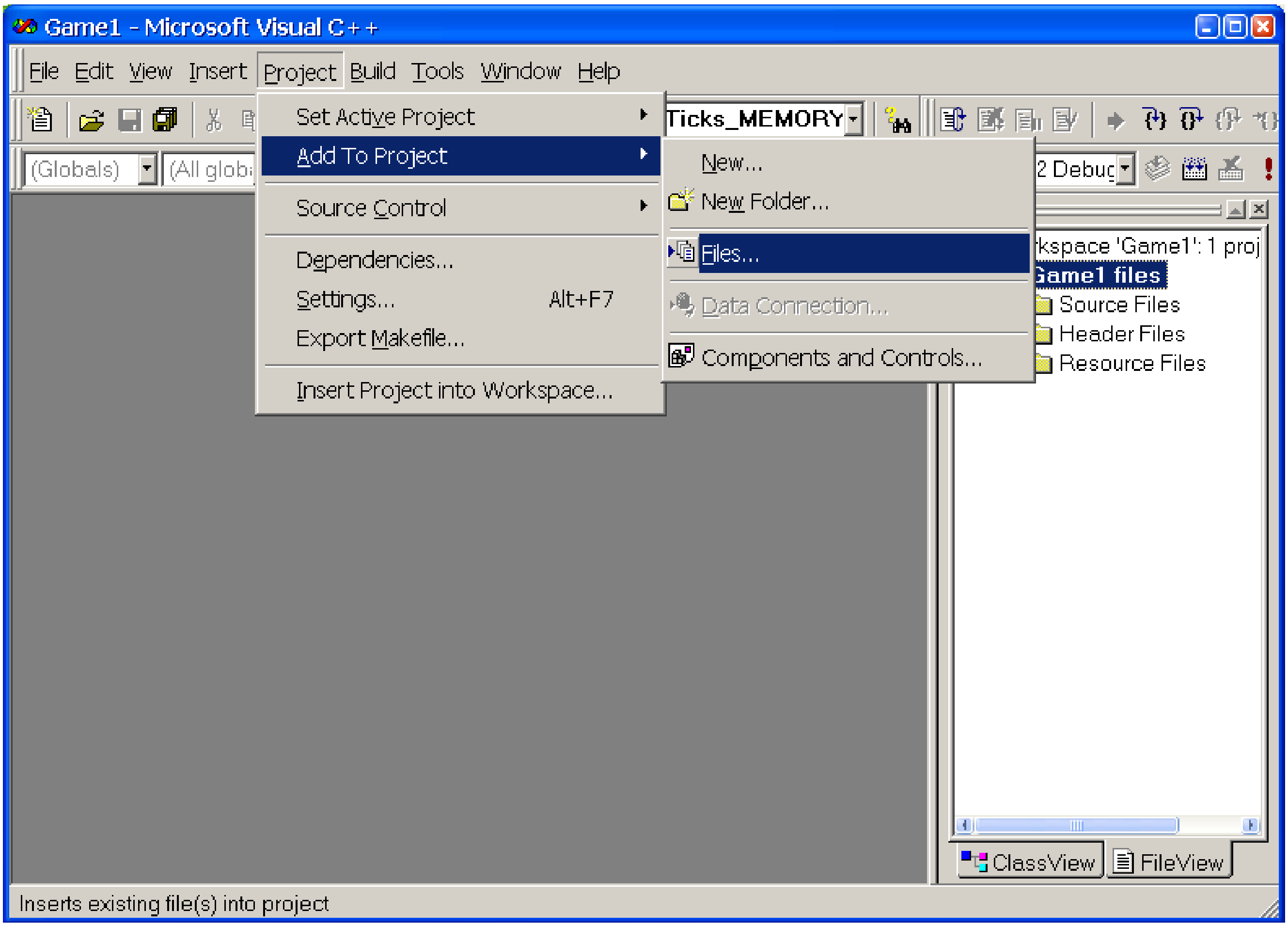
○ Use precompiled header file (.pch)

Project Options:

```
/nologo /MDd /W3 /Gm /GX /ZI /Od /D "WIN32" /D "_DEBUG"
/D "_CONSOLE" /D "_MBCS" /Fo"Debug/" /Fd"Debug/" /FD
/GZ  /c
```

3. Copy the DOSEmu.h and DOSEmu.cpp files to the project folder

4. Type the source code of the programs into files .CPP and .H, then add them to the project together with the DOSEmu.h DOSEmu.cpp files.



Inserts existing file(s) into project

Also you may just copy .PCX .C and .H files from the examples folder. Then rename the files .C->.CPP and add them together with .H files to the project.

5. Compile and run the project.

All ported book projects are in the workspace chapters_2_19.dsw

Open File-> Open Workspace-> Chapters_2_19.dsw, then make the project which you need active, compile and run it.

ADDITIONAL MATERIALS

MORE ABOUT ENGINES (BONUS)

Engine for the game Another World, Delphine Software, 1991

The book "Tricks of the Game-Programming Gurus" tell about the engines. The engineer is the constructor of engines (centers of localization of complexity).

Therefore, as the bonus I attached the story of the creation of one unique engine under DOS. In the late of 1980s, one engineer constructed a virtual machine, "twisting" byte code. The virtual machine has a lot of threading, and generates all the vector graphics of game screens, using only one function, drawing a colored polygon. The game on this engine is ported to a dozen platforms, the executable file takes 20 kilobytes.

The book contains two articles about the engine. One in Russian and one in English. Filling them is not duplicated, I recommend reading both.

The source code of the engine attached to the book in the ENGINES folder.

It is also available in the github repository

https://github.com/fabiensanglard/Another-World-Bytecode-Interpreter

The game Urban Chaos, Mucky Foot Productions, 1999

In 1999, the legendary game Urban Chaos was released on Windows, PlayStation and Dreamcast platforms.

It is a game for all times, combining the storyline, the decoration, the program engine as art.

The engine is such that, after 20 years Urban Chaos, does not lose to modern games on the graphic, and modern graphics cards make the game graphics only better.

As part of the work on Urban Chaos, a virtual machine was written that runs scripts in the dialect of the BASIC language.

The source code of the game attached to the book in the ENGINES folder.

It is also available in the github repository

https://github.com/dizzy2003/MuckyFoot-UrbanChaos

APPENDIX / ПРИЛОЖЕНИЕ

CHAPTERS / ГЛАВЫ

CHAP_02

SETMODEC.CPP

```
//------------------------------------------
// DOS EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------

#include <stdio.h>

#define VGA256 0x13
#define TEXT_MODE 0x03

extern void Set_Mode(int mode);


void main2(void)
{

// set video mode to 320x200 256 color mode

Set_Mode(VGA256);

// wait for keyboard to be hit

while(!kbhit()){ FAST_CPU_WAIT(10); }

// go back to text mode

Set_Mode(TEXT_MODE);

      } // end main
```

FILLC.CPP

```
//---------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//---------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//---------------------------------------

#include <stdio.h>
```

```
#define VGA256 0x13
#define TEXT_MODE 0x03

extern void Set_Mode(int mode);

void Fill_Screen(int color)
{
      char far * screen_ram = (char far *)MEMORY_0xA0000000; // vram byte ptr
      memset(screen_ram,color,320*200);                     // clear buffer

      _redraw_screen();
}


void main2(void)
{
int t;

// set video mode to 320x200 256 color mode

Set_Mode(VGA256);

// fill the screen with 1's which in the defualt pallete will be blue


for (t=0; t<1000; t++)
Fill_Screen(t);

// wait for keyboard to be hit

// while(!kbhit()){}

// go back to text mode

Set_Mode(TEXT_MODE);

} // end main
```

CHAP_03

KEY.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <dos.h>
//#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
//#include <graph.h>


// D E F I N E S  ////////////////////////////////////////////////////////////

// bitmasks for control keys/shift status

#define SHIFT_R             0x0001
#define SHIFT_L             0x0002
```

```
#define CTRL                0x0004
#define ALT                 0x0008
#define SCROLL_LOCK_ON      0x0010
#define NUM_LOCK_ON         0x0020
#define CAPS_LOCK_ON        0x0040
#define INSERT_MODE         0x0080
#define CTRL_L              0x0100
#define ALT_L               0x0200
#define CTRL_R              0x0400
#define ALT_R               0x0800
#define SCROLL_LOCK_DWN     0x1000
#define NUM_LOCK_DWN        0x2000
#define CAPS_LOCK_DWN       0x4000
#define SYS_REQ_DWN         0x8000

// scan code values, note keys with two symbols on them are the same so I will
// consistantly try to use the lower symbol for example. the 1 key also has a
// ! above it, but we would just call it the SCAN_1 key.

#define SCAN_ESC          1
#define SCAN_1            2
#define SCAN_2            3
#define SCAN_3            4
#define SCAN_4            5
#define SCAN_5            6
#define SCAN_6            7
#define SCAN_7            8
#define SCAN_8            9
#define SCAN_9            10
#define SCAN_0            11
#define SCAN_MINUS        12
#define SCAN_EQUALS       13
#define SCAN_BKSP         14
#define SCAN_TAB          15
#define SCAN_Q            16
#define SCAN_W            17
#define SCAN_E            18
#define SCAN_R            19
#define SCAN_T            20
#define SCAN_Y            21
#define SCAN_U            22
#define SCAN_I            23
#define SCAN_O            24
#define SCAN_P            25
#define SCAN_LFT_BRACKET  26
#define SCAN_RGT_BRACKET  27
#define SCAN_ENTER        28
#define SCAN_CTRL         29

#define SCAN_A            30
#define SCAN_S            31
#define SCAN_D            32
#define SCAN_F            33
#define SCAN_G            34
#define SCAN_H            35
#define SCAN_J            36
#define SCAN_K            37
#define SCAN_L            38

#define SCAN_SEMI         39
#define SCAN_APOS         40
#define SCAN_TILDE        41

#define SCAN_LEFT_SHIFT   42
#define SCAN_BACK_SLASH   43
#define SCAN_Z            44
#define SCAN_X            45
#define SCAN_C            46
#define SCAN_V            47
#define SCAN_B            48
#define SCAN_N            49
```

```
#define SCAN_M            50
#define SCAN_COMMA        51
#define SCAN_PERIOD       52
#define SCAN_FOWARD_SLASH 53
#define SCAN_RIGHT_SHIFT  54
#define SCAN_PRT_SCRN     55
#define SCAN_ALT          56
#define SCAN_SPACE        57
#define SCAN_CAPS_LOCK    58
#define SCAN_F1           59
#define SCAN_F2           60
#define SCAN_F3           61
#define SCAN_F4           62
#define SCAN_F5           63
#define SCAN_F6           64
#define SCAN_F7           65
#define SCAN_F8           66
#define SCAN_F9           67
#define SCAN_F10          68
#define SCAN_F11          133
#define SCAN_F12          134
#define SCAN_NUM_LOCK     69
#define SCAN_SCROLL_LOCK  70
#define SCAN_HOME         71
#define SCAN_UP           72
#define SCAN_PGUP         73
#define SCAN_NUM_MINUS    74
#define SCAN_LEFT         75
#define SCAN_CENTER       76
#define SCAN_RIGHT        77
#define SCAN_NUM_PLUS     78
#define SCAN_END          79
#define SCAN_DOWN         80
#define SCAN_PGDWN        81
#define SCAN_INS          82
#define SCAN_DEL          83

// F U N C T I O N S /////////////////////////////////////////////////////////

unsigned char Get_Ascii_Key(void)

{

// if there is a normal ascii key waiting then return it, else return 0

if (_bios_keybrd(_KEYBRD_READY))
 return(_bios_keybrd(_KEYBRD_READ));
else return(0);

} // end Get_Ascii_Key

//////////////////////////////////////////////////////////////////////////////

unsigned int Get_Control_Keys(unsigned int mask)
{
// return the status of all the requested control key

       return(mask & _bios_keybrd(_KEYBRD_SHIFTSTATUS));

} // end Get_Control_Keys

//////////////////////////////////////////////////////////////////////////////

unsigned char Get_Scan_Code(void)
{
// get the scan code of a key press, since we have to look at status bits
// let's use the inline assembler


// is a key ready?
```

```
//------------------------------------------
// NO ANY ASM
//------------------------------------------
/*

__asm
    {
    mov ah,01h          ; function 1: is a key ready?
    int 16h             ; call the interrupt
    jz empty            ; there was no key so exit
    mov ah,00h          ; function 0: get the scan code please
    int 16h             ; call the interrupt
    mov al,ah           ; result was in ah so put into al
    xor ah,ah           ; zero out ah
    jmp done            ; data's in ax...let's blaze!

empty:
     xor ax,ax          ; clear out ax i.e. 0 means no key
done:

    } // end asm
*/
//------------------------------------------


      REGS r;

      r.h.ah = 0x01;
      int86(0x16, &r, &r);
      if(r.w.cflag & I386_FLAG_ZF) return 0;

      r.h.ah = 0x00;
      int86(0x16, &r, &r);
      r.h.al = r.h.ah;
      r.h.ah ^= r.h.ah;

      return r.w.ax;

// since data is in ax it will be returned properly

} // end Get_Scan_Code

//////////////////////////////////////////////////////////////////////////////

void main2(void) // keyboard demo
{
unsigned char key;
int done=0;
unsigned int control;

_clearscreen(_GCLEARSCREEN);

while(!done)
     {

     _settextposition(2,0);

     if ( (key = Get_Scan_Code()) )
        printf("%c %d  ",key,key);

     // test for ctrl and alt keys

     if (Get_Control_Keys(CTRL))
        printf("\ncontrol key pressed");

     if (Get_Control_Keys(ALT))
        printf("\nalt key pressed    ");

     if (key==16) done=1; // 16 is the scan code for 'q'

       if(!key) printf("               \n                        ");
     } // end main

} // end main
```

MOUSE.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <dos.h>
//#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
//#include <graph.h>


// D E F I N E S  ////////////////////////////////////////////////////////////

// mouse sub-function calls

#define MOUSE_INT                0x33 //mouse interrupt number
#define MOUSE_RESET              0x00 // reset the mouse
#define MOUSE_SHOW               0x01 // show the mouse
#define MOUSE_HIDE               0x02 // hide the mouse
#define MOUSE_BUTT_POS           0x03 // get buttons and postion
#define MOUSE_SET_SENSITIVITY    0x1A // set the sensitivity of mouse 0-100
#define MOUSE_MOTION_REL         0x0B // query motion counters to compute
                                      // relative motion

// defines to make reading buttons easier

#define MOUSE_LEFT_BUTTON        0x01 // left button mask
#define MOUSE_RIGHT_BUTTON       0x02 // right button mask
#define MOUSE_CENTER_BUTTON      0x04 // center button mask


// G L O B A L S  ////////////////////////////////////////////////////////////



// F U N C T I O N S /////////////////////////////////////////////////////////


int Squeeze_Mouse(int command, int *x, int *y,int *buttons)
{
// mouse interface, we'll use _int86 instead of inline asm...Why? No real reason.
// what function is caller requesting?

union _REGS inregs, outregs;

switch(command)
      {

      case MOUSE_RESET:
           {

           inregs.x.ax = 0x00; // subfunction 0: reset
           _int86(MOUSE_INT, &inregs, &outregs);
```

```
     *buttons = outregs.x.bx; // return number of buttons
     return(outregs.x.ax);    // return overall success/failure

     } break;

case MOUSE_SHOW:
     {
     // this function increments the internal show mouse counter.
     // when it is equal to 0 then the mouse will be displayed.

     inregs.x.ax = 0x01; // subfunction 1: increment show flag
     _int86(MOUSE_INT, &inregs, &outregs);

     return(1);

     } break;

case MOUSE_HIDE:
     {
     // this function decrements the internal show mouse counter.
     // when it is equal to -1 then the mouse will be hidden.

     inregs.x.ax = 0x02; // subfunction 2: decrement show flag
     _int86(MOUSE_INT, &inregs, &outregs);

     return(1);

     } break;

case MOUSE_BUTT_POS:
     {
     // this functions gets the buttons and returns the absolute mouse
     // positions in the vars x,y, and buttons, respectively

     inregs.x.ax = 0x03; // subfunction 3: get position and buttons
     _int86(MOUSE_INT, &inregs, &outregs);

     // extract the info and send back to caller via pointers
     *x       = outregs.x.cx;
     *y       = outregs.x.dx;
     *buttons = outregs.x.bx;

     return(1);

     } break;

case MOUSE_MOTION_REL:
     {

     // this functions gets the relative mouse motions from the last
     // call and puts them in the vars x,y respectively

     inregs.x.ax = 0x03; // subfunction 11: get relative motion
     _int86(MOUSE_INT, &inregs, &outregs);

     // extract the info and send back to caller via pointers
     *x       = outregs.x.cx;
     *y       = outregs.x.dx;

     return(1);

     } break;

case MOUSE_SET_SENSITIVITY:
     {
     // this function sets the overall "sensitivity" of the mouse.
     // each axis can have a sensitivity from 1-100.  So the caller
     // should put 1-100 in both "x" and "y" before calling/
     // also "buttons" is used to send in the doublespeed value which
     // ranges from 1-100 also.
```

```
           inregs.x.bx = *x;
           inregs.x.cx = *y;
           inregs.x.dx = *buttons;

           inregs.x.ax = 0x1A; // subfunction 26: set sensitivity
           _int86(MOUSE_INT, &inregs, &outregs);

           return(1);

           } break;

      default:break;

      } // end switch

} // end Squeze_Mouse

///////////////////////////////////////////////////////////////////////////////

void main2(void)
{

int x,y,buttons,num_buttons;
int color=1;

// put the computer into graphics mode

_setvideomode(_VRES16COLOR); //  640x480 in 16 colors

// initialize mouse

Squeeze_Mouse(MOUSE_RESET,NULL,NULL,&num_buttons);

// show the mouse

Squeeze_Mouse(MOUSE_SHOW,NULL,NULL,NULL);

while(!kbhit())
     {
      FAST_CPU_WAIT(1);

     _settextposition(2,0);

     Squeeze_Mouse(MOUSE_BUTT_POS,&x,&y,&buttons);

     printf("mouse x=%d y=%d buttons=%d    ",x,y,buttons);

     // video easel

     if (buttons==1)
        {
        _setcolor(color);
        _setpixel(x-1,y-2);
        _setpixel(x,y-2);
        _setpixel(x-1,y-1);
        _setpixel(x,y-1);
        } // end if draw on

     if (buttons==2)
        {
        if (++color>15) color=0;

        // wait for mouse release

        while(buttons==2)
             {
             Squeeze_Mouse(MOUSE_BUTT_POS,&x,&y,&buttons);
             } // end while

        } // end if draw on
     } // end while

// place the computer back into text mode

_setvideomode(_DEFAULTMODE);

} // end main
```

CHAP_04

POINTY.CPP

```
//-----------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//-----------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//-----------------------------------------


#include <stdio.h> // include the basics
// #include <graph.h> // include Microsofts Graphics Header

void main2(void)
{

int x,y,index,color;

// put the computer into graphics mode

_setvideomode(_VRES16COLOR); //  640x480 in 16 colors

// let's draw 100 points randomly on the screen

for (index = 0; index<10000; index++)
     {
     // get a random position and color and plot a point there

     x = rand()%640;
     y = rand()%480;
     color = rand()%16;
    _setcolor(color);  // set the color of the pixel to be drawn
    _setpixel(x,y);    // draw the pixel

     } // end for index

// wait for the user to hit a key

while(!kbhit()){ FAST_CPU_WAIT(10); }

// place the computer back into text mode

_setvideomode(_DEFAULTMODE);

} // end main
```

LINER.CPP

```
//-----------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//-----------------------------------------
```

```
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


#include <stdio.h> // include the basics
// #include <graph.h>// include Microsofts Graphics Header

void main2(void)
{

int x1,y1,x2,y2,color,index;

// put the computer into graphics mode

_setvideomode(_VRES16COLOR); //  640x480 in 16 colors

// let's draw 1,000 lines randomly on the screen

for (index = 0; index<1000; index++)
     {
     // get a random positions and color and draw a line there

     x1 = rand()%640;  //  x of starting point
     y1 = rand()%480;  //  y of starting point
     x2 = rand()%640;  //  x of ending point
     y2 = rand()%480;  //  y of ending point
     color = rand()%16;
    _setcolor(color);  // set the color of the pixel to be drawn
    _moveto(x1,y1);    // move to the start of the line
    _lineto(x2,y2);    //  draw the line

     } // end for index

// wait for the user to hit a key

while(!kbhit()){ FAST_CPU_WAIT(10); }

// place the computer back into text mode

_setvideomode(_DEFAULTMODE);

} // end main
```

POLYDRAW.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


#include <stdio.h> // include the basics
//#include <graph.h>// include Microsofts Graphics Header

void main2(void)
{

// put the computer into graphics mode
_setvideomode(_VRES16COLOR); //  640x480 in 16 colors

// draw a simple polygon
_setcolor(1);  // blue
_moveto(100,100); // vertex 1
_lineto(120,120);   // vertex 2
_lineto(150,200);   // vertex 3
_lineto(80,190);    // vertex 4
_lineto(80,60);      // vertex 5
_lineto(100,100);   // back to vertex 1 to close up the polygon

// now highlight each vertex in white

_setcolor(15);  // white
_setpixel(100,100); // vertex 1
_setpixel(120,120); // vertex 2
_setpixel(150,200); // vertex 3
_setpixel(80,190);  // vertex 4
_setpixel(80,60);   // vertex 5


// wait for the user to hit a key
while(!kbhit()){ FAST_CPU_WAIT(10); }

// place the computer back into text mode

_setvideomode(_DEFAULTMODE);

} // end main
```

FIELD.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <stdio.h> // include the basics
//#include <graph.h> // include Microsofts Graphics Header
#include <math.h>  // include math stuff

// D E F I N E S  ///////////////////////////////////////////////////////////

#define NUM_ASTEROIDS 10
#define ERASE 0
#define DRAW  1

// T Y P E D E F S ///////////////////////////////////////////////////////////

// the structure for a vertex

typedef struct vertex_typ
           {
           float x,y; // a single point in the 2-D plane.
           } vertex, *vertex_ptr;


// the structure for an object

typedef struct object_typ
           {
           int num_vertices;     // number of vertices in this object
           int color;            // color of object
           float xo,yo;          // position of object
           float x_velocity;     // x velocity of object
           float y_velocity;     // y velocity of object
           float scale;          // scale factor
           float angle;          // rotation rate
           vertex vertices[16];  // 16 vertices
           } object, *object_ptr;
```

```
// G L O B A L S ////////////////////////////////////////////////////////


object asteroids[NUM_ASTEROIDS];


// F U N C T I O N S ////////////////////////////////////////////////////

void Delay(int t)
{
       Sleep(t);
}

/*
void Delay(int t)
{

// take up some compute cycles

float x = 1;

while(t-->0)
     x=cos(x);


} // end Delay
*/

//////////////////////////////////////////////////////////////////////////////

void Scale_Object(object_ptr object,float scale)
{
int index;

// for all vertices scale the x and y component

for (index = 0; index<object->num_vertices; index++)
      {
      object->vertices[index].x *= scale;
      object->vertices[index].y *= scale;
      } // end for index

} // end Scale_Object

//////////////////////////////////////////////////////////////////////////////

void Rotate_Object(object_ptr object, float angle)
{
int index;
float x_new, y_new,cs,sn;

// pre-compute sin and cos
cs = cos(angle);
sn = sin(angle);

// for each vertex rotate it by angle
for (index=0; index<object->num_vertices; index++)
      {
       // rotate the vertex
      x_new  = object->vertices[index].x * cs -  object->vertices[index].y * sn;
      y_new  = object->vertices[index].y * cs + object->vertices[index].x * sn;

      // store the rotated vertex back into structure
      object->vertices[index].x = x_new;
      object->vertices[index].y = y_new;

      } // end for index

} // end Rotate_Object
```

```
/////////////////////////////////////////////////////////////////////////////

void Create_Field(void)
{

int index;

for (index=0; index<NUM_ASTEROIDS; index++)
    {

    // fill in the fields

    asteroids[index].num_vertices = 6;
    asteroids[index].color = 1  + rand() % 14; // always visable
    asteroids[index].xo    = 41 + rand() % 599;
    asteroids[index].yo    = 41 + rand() % 439;

    asteroids[index].x_velocity = -10 + rand() % 20;
    asteroids[index].y_velocity = -10 + rand() % 20;
    asteroids[index].scale      = (float)(rand() % 30) / 10;
    asteroids[index].angle      = (float)(- 50 + (float)(rand() % 100)) / 100;

    asteroids[index].vertices[0].x = 4.0;
    asteroids[index].vertices[0].y = 3.5;
    asteroids[index].vertices[1].x = 8.5;
    asteroids[index].vertices[1].y = -3.0;
    asteroids[index].vertices[2].x = 6;
    asteroids[index].vertices[2].y = -5;
    asteroids[index].vertices[3].x = 2;
    asteroids[index].vertices[3].y = -3;
    asteroids[index].vertices[4].x = -4;
    asteroids[index].vertices[4].y = -6;
    asteroids[index].vertices[5].x = -3.5;
    asteroids[index].vertices[5].y = 5.5;

    // now scale the asteroid to proper size

    Scale_Object((object_ptr)&asteroids[index], asteroids[index].scale);

    } // end for index

} // end Create_Field

/////////////////////////////////////////////////////////////////////////////

void Draw_Asteroids(int erase)
{

int index,vertex;
float xo,yo;

for (index=0; index<NUM_ASTEROIDS; index++)
    {

    // draw the asteroid

    if (erase==ERASE)
       _setcolor(0);
    else
       _setcolor(asteroids[index].color);

    // get position of object
    xo = asteroids[index].xo;
    yo = asteroids[index].yo;


    // moveto first vertex


_moveto((int)(xo+asteroids[index].vertices[0].x),(int)(yo+asteroids[index].vertices[0].y));
```

```
    for (vertex=1; vertex<asteroids[index].num_vertices; vertex++)
        {

_lineto((int)(xo+asteroids[index].vertices[vertex].x),(int)(yo+asteroids[index].vertices[ve
rtex].y));

        } // end for vertex

    // close object


_lineto((int)(xo+asteroids[index].vertices[0].x),(int)(yo+asteroids[index].vertices[0].y));

    } // end for index

} // end Draw_Asteroids

//////////////////////////////////////////////////////////////////////////////

void Translate_Asteroids()
{

int index;

for (index=0; index<NUM_ASTEROIDS; index++)
    {
    // translate current asteroid

    asteroids[index].xo += asteroids[index].x_velocity;
    asteroids[index].yo += asteroids[index].y_velocity;

    // collision detection i.e. bounds check

    if (asteroids[index].xo > 600 || asteroids[index].xo < 40)
        {
        asteroids[index].x_velocity = -asteroids[index].x_velocity;
        asteroids[index].xo += asteroids[index].x_velocity;
        }

    if (asteroids[index].yo > 440 || asteroids[index].yo < 40)
        {
        asteroids[index].y_velocity = -asteroids[index].y_velocity;
        asteroids[index].yo += asteroids[index].y_velocity;
        }

    } // end for index

} // end Translate_Asteroids

//////////////////////////////////////////////////////////////////////////////

void Rotate_Asteroids(void)
{

int index;

for (index=0; index<NUM_ASTEROIDS; index++)
    {
    // rotate current asteroid
    Rotate_Object((object_ptr)&asteroids[index], asteroids[index].angle);

    } // end for index

} // end Rotate_Asteroids


//////////////////////////////////////////////////////////////////////////////

void main2(void)
{
```

```
// put the computer into graphics mode
_setvideomode(_VRES16COLOR); //  640x480 in 16 colors

// initialize
Create_Field();

_set_render_options(TRUE, RENDER_MANUAL_REDRAW);

while(!kbhit())
     {
     // erase field

     Draw_Asteroids(ERASE);

     // transform field

     Rotate_Asteroids();

     Translate_Asteroids();

     // draw field

     Draw_Asteroids(DRAW);
        _redraw_screen();

     // wait a bit since we aren't syncing or double buffering...nuff said

     Delay(70);


     } // end while

// place the computer back into text mode

_setvideomode(_DEFAULTMODE);

} // end main
```

FIELD_DL.CPP

```
//-----------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//-----------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//-----------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <stdio.h> // include the basics
//#include <graph.h> // include Microsofts Graphics Header
#include <math.h>  // include math stuff

// D E F I N E S   //////////////////////////////////////////////////////////

#define NUM_ASTEROIDS 10
#define ERASE 0
#define DRAW  1

#define X_COMP 0
#define Y_COMP  1
#define N_COMP 2
```

```
// T Y P E D E F S ///////////////////////////////////////////////////////////

// new and improved vertex

typedef struct vertex_typ
           {
           float p[3];  // a single point in the 2-D plane with normalizing factor
           } vertex, *vertex_ptr;

// a general matrix structure

typedef struct matrix_typ
           {
           float elem[3][3];  // storage for a 3x3 martrix
           } matrix, *matrix_ptr;


// the structure for an object

typedef struct object_typ
           {
           int num_vertices;      // number of vertices in this object
           int color;             // color of object
           float xo,yo;           // position of object
           float x_velocity;      // x velocity of object
           float y_velocity;      // y velocity of object
           matrix scale;          // the object scaling matrix
           matrix rotation;       // the objects rotation and translation matrix

           vertex vertices[16];  // 16 vertices
           } object, *object_ptr;


// G L O B A L S /////////////////////////////////////////////////////////////


object asteroids[NUM_ASTEROIDS];


// F U N C T I O N S /////////////////////////////////////////////////////////

void Delay(int t)
{
      Sleep(t);
}

/*
void Delay(int t)
{

// take up some compute cycles

float x = 1;

while(t-->0)
     x=cos(x);


} // end Delay
*/

//////////////////////////////////////////////////////////////////////////////

void Make_Identity(matrix_ptr i)
{

// makes the sent matrix into an identity matrix

i->elem[0][0] = i->elem[1][1] = i->elem[2][2] = 1;
i->elem[0][1] = i->elem[1][0] = i->elem[1][2] = 0;
i->elem[2][0] = i->elem[0][2] = i->elem[2][1] = 0;
```

```
} // end Make_Identity

//////////////////////////////////////////////////////////////////////////////

void Clear_Matrix(matrix_ptr m)
{

// zeros out the sent matrix

m->elem[0][0] = m->elem[1][1] = m->elem[2][2] = 0;
m->elem[0][1] = m->elem[1][0] = m->elem[1][2] = 0;
m->elem[2][0] = m->elem[0][2] = m->elem[2][1] = 0;


} // end Clear_Matrix

//////////////////////////////////////////////////////////////////////////////


void Mat_Mul(vertex_ptr v,matrix_ptr m)
{

// do a multiplication of a 1x3 * 3x3 the result is again a 1x3
// for speed manually do the multiplication by specifying each multiplication
// and addition manually (apprentice trick)

float x_new, y_new;

x_new = v->p[0]*m->elem[0][0] + v->p[1]*m->elem[1][0] + m->elem[2][0];
y_new = v->p[0]*m->elem[0][1] + v->p[1]*m->elem[1][1] + m->elem[2][1];

v->p[X_COMP] = x_new;
v->p[Y_COMP] = y_new;

// note we need not change N_COMP since it is always 1

} // end Mat_Mul

//////////////////////////////////////////////////////////////////////////////


void Scale_Object_Mat(object_ptr obj)
{

int index;

// scale the object, just multiply each point in the object by it's scaling
// matrix

for (index=0; index<obj->num_vertices; index++)
    {

    Mat_Mul((vertex_ptr)&obj->vertices[index],(matrix_ptr)&obj->scale);

    } // end for index

} // end Scale_Oject_Mat


//////////////////////////////////////////////////////////////////////////////

Rotate_Object_Mat(object_ptr obj)
{

int index;

// rotate the object, just multiply each point in the object by it's rotation
// matrix
```

```
for (index=0; index<obj->num_vertices; index++)
    {

    Mat_Mul((vertex_ptr)&obj->vertices[index],(matrix_ptr)&obj->rotation);

    } // end for index

} // end Rotate_Oject_Mat


//////////////////////////////////////////////////////////////////////////////

void Create_Field(void)
{

int index;
float angle,c,s;

// this function creates the asteroid field

for (index=0; index<NUM_ASTEROIDS; index++)
    {

    // fill in the fields

    asteroids[index].num_vertices = 6;
    asteroids[index].color = 1  + rand() % 14; // always visable
    asteroids[index].xo    = 41 + rand() % 599;
    asteroids[index].yo    = 41 + rand() % 439;
    asteroids[index].x_velocity = -10 + rand() % 20;
    asteroids[index].y_velocity = -10 + rand() % 20;

    // clear out matrix

    Make_Identity((matrix_ptr)&asteroids[index].rotation);

    // now setup up rotation  matrix

    angle = (float)(- 50 + (float)(rand() % 100)) / 100;

    c=cos(angle);
    s=sin(angle);

    asteroids[index].rotation.elem[0][0] = c;
    asteroids[index].rotation.elem[0][1] = -s;
    asteroids[index].rotation.elem[1][0] = s;
    asteroids[index].rotation.elem[1][1] = c;

    // set up scaling  matrix

    // clear out matrix

    Make_Identity((matrix_ptr)&asteroids[index].scale);

    asteroids[index].scale.elem[0][0] = (float)(rand() % 30) / 10;
    asteroids[index].scale.elem[1][1] = asteroids[index].scale.elem[0][0];


    asteroids[index].vertices[0].p[X_COMP] = 4.0;
    asteroids[index].vertices[0].p[Y_COMP] = 3.5;
    asteroids[index].vertices[0].p[N_COMP] = 1;

    asteroids[index].vertices[1].p[X_COMP] = 8.5;
    asteroids[index].vertices[1].p[Y_COMP] = -3.0;
    asteroids[index].vertices[1].p[N_COMP] = 1;

    asteroids[index].vertices[2].p[X_COMP] = 6;
    asteroids[index].vertices[2].p[Y_COMP] = -5;
    asteroids[index].vertices[2].p[N_COMP] = 1;

    asteroids[index].vertices[3].p[X_COMP] = 2;
    asteroids[index].vertices[3].p[Y_COMP] = -3;
    asteroids[index].vertices[3].p[N_COMP] = 1;

    asteroids[index].vertices[4].p[X_COMP] = -4;
    asteroids[index].vertices[4].p[Y_COMP] = -6;
    asteroids[index].vertices[4].p[N_COMP] = 1;

    asteroids[index].vertices[5].p[X_COMP] = -3.5;
    asteroids[index].vertices[5].p[Y_COMP] = 5.5;
    asteroids[index].vertices[5].p[N_COMP] = 1;

    // now scale the asteroid to proper size

    Scale_Object_Mat((object_ptr)&asteroids[index]);

    } // end for index

} // end Create_Field

//////////////////////////////////////////////////////////////////////////////

void Draw_Asteroids(int erase)
{

int index,vertex;
float xo,yo;

// this function draws the asteroids or erases them depending on the sent flag

for (index=0; index<NUM_ASTEROIDS; index++)
    {

    // draw the asteroid

    if (erase==ERASE)
       _setcolor(0);
    else
       _setcolor(asteroids[index].color);

    // get position of object
    xo = asteroids[index].xo;
    yo = asteroids[index].yo;


    // moveto first vertex

    _moveto((int)(xo+asteroids[index].vertices[0].p[X_COMP]),
            (int)(yo+asteroids[index].vertices[0].p[Y_COMP]));

    for (vertex=1; vertex<asteroids[index].num_vertices; vertex++)
        {
        _lineto((int)(xo+asteroids[index].vertices[vertex].p[X_COMP]),
                (int)(yo+asteroids[index].vertices[vertex].p[Y_COMP]));

        } // end for vertex

    // close object

    _lineto((int)(xo+asteroids[index].vertices[0].p[X_COMP]),
            (int)(yo+asteroids[index].vertices[0].p[Y_COMP]));


    } // end for index

} // end Draw_Asteroids

//////////////////////////////////////////////////////////////////////////////

void Translate_Asteroids(void)
{
```

```
int index;

// this function moves the asteroids

for (index=0; index<NUM_ASTEROIDS; index++)
    {
    // translate current asteroid

    asteroids[index].xo += asteroids[index].x_velocity;
    asteroids[index].yo += asteroids[index].y_velocity;

    // collision detection i.e. bounds check

    if (asteroids[index].xo > 600 || asteroids[index].xo < 40)
        {
        asteroids[index].x_velocity = -asteroids[index].x_velocity;
        asteroids[index].xo += asteroids[index].x_velocity;
        }

    if (asteroids[index].yo > 440 || asteroids[index].yo < 40)
        {
        asteroids[index].y_velocity = -asteroids[index].y_velocity;
        asteroids[index].yo += asteroids[index].y_velocity;
        }

    } // end for index

} // end Translate_Asteroids

//////////////////////////////////////////////////////////////////////////////

void Rotate_Asteroids()
{

int index;

for (index=0; index<NUM_ASTEROIDS; index++)
    {
    // rotate current asteroid
    Rotate_Object_Mat((object_ptr)&asteroids[index]);

    } // end for index

} // end Rotate_Asteroids


//////////////////////////////////////////////////////////////////////////////

void main2(void)
{

// put the computer into graphics mode
_setvideomode(_VRES16COLOR); //  640x480 in 16 colors

// initialize
Create_Field();

while(!kbhit())
     {
     // erase field

     Draw_Asteroids(ERASE);

     // transform field

     Rotate_Asteroids();

     Translate_Asteroids();

     // draw field
     Draw_Asteroids(DRAW);

     // wait a bit since we aren't syncing or double buffering...nuff said

     Delay(10);


     } // end while

// place the computer back into text mode

_setvideomode(_DEFAULTMODE);

} // end main
```

CHAP_05

PALDEMO.CPP

```
//-------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//-------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//-------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
//#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <math.h>
#include <string.h>

// D E F I N E S  ////////////////////////////////////////////////////////////

#define ROM_CHAR_SET_SEG 0xF000  // segment of 8x8 ROM character set
#define ROM_CHAR_SET_OFF 0xFA6E  // begining offset of 8x8 ROM character set

#define VGA256            0x13
#define TEXT_MODE         0x03

#define PALETTE_MASK      0x3c6

#define PALETTE_REGISTER_RD 0x3c7
#define PALETTE_REGISTER_WR 0x3c8

#define PALETTE_DATA      0x3c9

#define SCREEN_WIDTH      (unsigned int)320
#define SCREEN_HEIGHT     (unsigned int)200

// S T R U C T U R E S ///////////////////////////////////////////////////////

// this structure holds a RGB triple in three bytes

typedef struct RGB_color_typ
        {
```

```
        unsigned char red;    // red   component of color 0-63
        unsigned char green;  // green component of color 0-63
        unsigned char blue;   // blue  component of color 0-63

        } RGB_color, *RGB_color_ptr;

// E X T E R N A L S ////////////////////////////////////////////////////


extern void Set_Mode(int mode);


// P R O T O T Y P E S //////////////////////////////////////////////////

void Set_Palette_Register(int index, RGB_color_ptr color);

void Get_Palette_Register(int index, RGB_color_ptr color);

void Create_Cool_Palette();

void V_Line(int y1,int y2,int x,unsigned int color);

// G L O B A L S  ///////////////////////////////////////////////////////

unsigned char far *video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr
unsigned int far *video_buffer_w= (unsigned int far *)MEMORY_0xA0000000;  // vram word ptr

// F U N C T I O N S ////////////////////////////////////////////////////

void Set_Palette_Register(int index, RGB_color_ptr color)
{

// this function sets a single color look up table value indexed by index
// with the value in the color structure

// tell VGA card we are going to update a pallete register

_outp(PALETTE_MASK,0xff);

// tell vga card which register we will be updating

_outp(PALETTE_REGISTER_WR, index);

// now update the RGB triple, note the same port is used each time

_outp(PALETTE_DATA,color->red);
_outp(PALETTE_DATA,color->green);
_outp(PALETTE_DATA,color->blue);

} // end Set_Palette_Color

///////////////////////////////////////////////////////////////////////////////

void Get_Palette_Register(int index, RGB_color_ptr color)
{

// this function gets the data out of a color lookup regsiter and places it
// into color

// set the palette mask register

_outp(PALETTE_MASK,0xff);

// tell vga card which register we will be reading

_outp(PALETTE_REGISTER_RD, index);

// now extract the data

color->red   = _inp(PALETTE_DATA);
color->green = _inp(PALETTE_DATA);
```

```
color->blue  = _inp(PALETTE_DATA);

} // end Get_Palette_Color

//////////////////////////////////////////////////////////////////////////////

void Create_Cool_Palette(void)
{

// this function creates a cool palette. 64 shades of gray, 64 of red,
// 64 of green and finally 64 of blue.

RGB_color color;

int index;

// swip thru the color registers and create 4 banks of 64 colors

for (index=0; index < 64; index++)
    {

    // grays

    color.red   = index;
    color.green = index;
    color.blue  = index;

    Set_Palette_Register(index, (RGB_color_ptr)&color);

    // reds

    color.red   = index;
    color.green = 0;
    color.blue  = 0;

    Set_Palette_Register(index+64, (RGB_color_ptr)&color);

    // greens

    color.red   = 0;
    color.green = index;
    color.blue  = 0;

    Set_Palette_Register(index+128, (RGB_color_ptr)&color);

    // blues

    color.red   = 0;
    color.green = 0;
    color.blue  = index;

    Set_Palette_Register(index+192, (RGB_color_ptr)&color);

    } // end index


} // end Create_Cool_Palette

//////////////////////////////////////////////////////////////////////////////

void V_Line(int y1,int y2,int x,unsigned int color)
{
// draw a vertical line, note y2 > y1

unsigned int line_offset,
                    index;

// compute starting position

line_offset = ((y1<<8) + (y1<<6)) + x;
```

```
//color = rand()%256;

for (index=0; index<=y2-y1; index++)
    {
    video_buffer[line_offset] = color;

     line_offset+=320; // move to next line

    } // end for index

} // end V_Line

//M A I N ////////////////////////////////////////////////////////////

void main2(void)
{
int index;
RGB_color color,color_1;

// set video mode to 320x200 256 color mode

Set_Mode(VGA256);
video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr
video_buffer_w= (unsigned int far *)MEMORY_0xA0000000;  // vram word ptr

// THIS FOR SPEEDUP ONLY. YOU MAY COMMENT IT
_set_render_options(TRUE, RENDER_NOT_REDRAW_IF_BY_PORT_PALETTE_CHANGED);

// create the color palette

Create_Cool_Palette();

// draw a bunch of vertical lines, one for each color

for (index=0; index<320; index++)
    V_Line(0,199,index,index);
_redraw_screen();

// wait for user to hit a key

while(!kbhit())
     {
     Get_Palette_Register(0,(RGB_color_ptr)&color_1);

     for (index=0; index<=254; index++)
         {
         Get_Palette_Register(index+1,(RGB_color_ptr)&color);
         Set_Palette_Register(index,(RGB_color_ptr)&color);

// THIS FOR SPEEDUP ONLY. YOU MAY COMMENT IT
if(!(index % 20)) _redraw_screen();
         } // end for

         Set_Palette_Register(255,(RGB_color_ptr)&color_1);

// THIS FOR SPEEDUP ONLY. YOU MAY COMMENT IT
_redraw_screen();

     } // end while

// go back to text mode

Set_Mode(TEXT_MODE);

} // end main
```

TOMB.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
//#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <math.h>
#include <string.h>

// D E F I N E S  ////////////////////////////////////////////////////////////

#define ROM_CHAR_SET_SEG 0xF000  // segment of 8x8 ROM character set
#define ROM_CHAR_SET_OFF 0xFA6E  // begining offset of 8x8 ROM character set

#define VGA256            0x13
#define TEXT_MODE         0x03

#define PALETTE_MASK         0x3c6
#define PALETTE_REGISTER_RD  0x3c7
#define PALETTE_REGISTER_WR  0x3c8
#define PALETTE_DATA         0x3c9

#define SCREEN_WIDTH      (unsigned int)320
#define SCREEN_HEIGHT     (unsigned int)200

#define CHAR_WIDTH        8
#define CHAR_HEIGHT       8

#define SPRITE_WIDTH      24
#define SPRITE_HEIGHT     24
#define MAX_SPRITE_FRAMES 16
#define SPRITE_DEAD       0
#define SPRITE_ALIVE      1
#define SPRITE_DYING      2

// S T R U C T U R E S ///////////////////////////////////////////////////////

// this structure holds a RGB triple in three bytes

typedef struct RGB_color_typ
        {

        unsigned char red;    // red   component of color 0-63
        unsigned char green;  // green component of color 0-63
        unsigned char blue;   // blue  component of color 0-63

        } RGB_color, *RGB_color_ptr;

typedef struct pcx_header_typ
        {
        char manufacturer;
        char version;
        char encoding;
        char bits_per_pixel;
        int x,y;
        int width,height;
        int horz_res;
        int vert_res;
        char ega_palette[48];
        char reserved;
        char num_color_planes;
        int bytes_per_line;
        int palette_type;
        char padding[58];

        } pcx_header, *pcx_header_ptr;


typedef struct pcx_picture_typ
        {
        pcx_header header;
        RGB_color palette[256];
        char far *buffer;

        } pcx_picture, *pcx_picture_ptr;


typedef struct sprite_typ
        {
        int x,y;            // position of sprite
        int x_old,y_old;    // old position of sprite
        int width,height;   // dimensions of sprite in pixels
        int anim_clock;     // the animation clock
        int anim_speed;     // the animation speed
        int motion_speed;   // the motion speed
        int motion_clock;   // the motion clock

        char far *frames[MAX_SPRITE_FRAMES]; // array of pointers to the images
        int curr_frame;                      // current frame being displayed
        int num_frames;                      // total number of frames
        int state;                           // state of sprite, alive, dead...
        char far *background;                // whats under the sprite

        } sprite, *sprite_ptr;

// E X T E R N A L S //////////////////////////////////////////////////////


extern void Set_Mode(int mode);


// P R O T O T Y P E S ////////////////////////////////////////////////////

void Set_Palette_Register(int index, RGB_color_ptr color);

void Plot_Pixel_Fast(int x,int y,unsigned char color);

void PCX_Init(pcx_picture *image);

void PCX_Delete(pcx_picture *image);

void PCX_Load(char *filename, pcx_picture_ptr image,int enable_palette);

void PCX_Show_Buffer(pcx_picture_ptr image);

// G L O B A L S  /////////////////////////////////////////////////////////

unsigned char far *video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr
unsigned int far *video_buffer_w= (unsigned int far *)MEMORY_0xA0000000;  // vram word ptr
unsigned char far *rom_char_set = (unsigned char far *)MEMORY_0xF000FA6EL; // rom
characters 8x8

// F U N C T I O N S //////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////

void Blit_Char(int xc,int yc,char c,int color)
{
// this function uses the rom 8x8 character set to blit a character on the
// video screen, notice the trick used to extract bits out of each character
// byte that comprises a line

int offset,x,y;
unsigned char data;
char far *work_char;
unsigned char bit_mask = 0x80;

// compute starting offset in rom character lookup table

work_char = (char *)rom_char_set + c * CHAR_HEIGHT;

// compute offset of character in video buffer

offset = (yc << 8) + (yc << 6) + xc;

for (y=0; y<CHAR_HEIGHT; y++)
    {
    // reset bit mask

    bit_mask = 0x80;

    for (x=0; x<CHAR_WIDTH; x++)
        {
        // test for transparent pixel i.e. 0, if not transparent then draw

        if ((*work_char & bit_mask))
             video_buffer[offset+x] = color;

        // shift bit mask

        bit_mask = (bit_mask>>1);

        } // end for x

    // move to next line in video buffer and in rom character data area

    offset      += SCREEN_WIDTH;
    work_char++;

    } // end for y

_redraw_screen();
} // end Blit_Char

//////////////////////////////////////////////////////////////////////////////

void Blit_String(int x,int y,int color, char *string)
{
// this function blits an entire string on the screen with fixed spacing
// between each character.  it calls blit_char.

int index;

for (index=0; string[index]!=0; index++)
     {

     Blit_Char(x+(index<<3),y,string[index],color);

     } /* end while */

} /* end Blit_String */

//////////////////////////////////////////////////////////////////////////////

void Delay(int t)
{

float x = 1;

while(t-->0)
```

```
     x=cos(x);

} // end Delay

//////////////////////////////////////////////////////////////////////////////

void Set_Palette_Register(int index, RGB_color_ptr color)
{

// this function sets a single color look up table value indexed by index
// with the value in the color structure

// tell VGA card we are going to update a pallete register

_outp(PALETTE_MASK,0xff);

// tell vga card which register we will be updating

_outp(PALETTE_REGISTER_WR, index);

// now update the RGB triple, note the same port is used each time

_outp(PALETTE_DATA,color->red);
_outp(PALETTE_DATA,color->green);
_outp(PALETTE_DATA,color->blue);

} // end Set_Palette_Color

//////////////////////////////////////////////////////////////////////////////

void PCX_Init(pcx_picture_ptr image)
{
// this function allocates the buffer region needed to load a pcx file

if (!(image->buffer = (char far *)malloc(SCREEN_WIDTH * SCREEN_HEIGHT + 1)))

   printf("\ncouldn't allocate screen buffer");

} // end PCX_Init

//////////////////////////////////////////////////////////////////////////////

void Plot_Pixel_Fast(int x,int y,unsigned char color)
{

// plots the pixel in the desired color a little quicker using binary shifting
// to accomplish the multiplications

// use the fact that 320*y = 256*y + 64*y = y<<8 + y<<6

video_buffer[((y<<8) + (y<<6)) + x] = color;
} // end Plot_Pixel_Fast

//////////////////////////////////////////////////////////////////////////////

void PCX_Delete(pcx_picture_ptr image)
{
// this function de-allocates the buffer region used for the pcx file load

_ffree(image->buffer);

} // end PCX_Delete

//////////////////////////////////////////////////////////////////////////////

void PCX_Load(char *filename, pcx_picture_ptr image,int enable_palette)
{
// this function loads a pcx file into a picture structure, the actual image
// data for the pcx file is decompressed and expanded into a secondary buffer
// within the picture structure, the separate images can be grabbed from this
// buffer later.  also the header and palette are loaded
```

```
FILE *fp /*, *fopen() */;
int num_bytes,index;
long count;
unsigned char data;
char far *temp_buffer;

// open the file

fp = fopen(filename,"rb");

// load the header

temp_buffer = (char far *)image;

for (index=0; index<128; index++)
    {
    temp_buffer[index] = getc(fp);
    } // end for index

// load the data and decompress into buffer

count=0;

while(count<=SCREEN_WIDTH * SCREEN_HEIGHT)
     {
     // get the first piece of data

     data = getc(fp);

     // is this a rle?

     if (data>=192 && data<=255)
        {
        // how many bytes in run?

        num_bytes = data-192;

        // get the actual data for the run

        data  = getc(fp);

        // replicate data in buffer num_bytes times

        while(num_bytes-->0)
             {
             image->buffer[count++] = data;

             } // end while

        } // end if rle
     else
        {
        // actual data, just copy it into buffer at next location

        image->buffer[count++] = data;

        } // end else not rle

     } // end while

// move to end of file then back up 768 bytes i.e. to begining of palette

fseek(fp,-768L,SEEK_END);

// load the pallete into the palette

for (index=0; index<256; index++)
    {
    // get the red component
    image->palette[index].red   = (getc(fp) >> 2);

    // get the green component

    image->palette[index].green = (getc(fp) >> 2);

    // get the blue component

    image->palette[index].blue  = (getc(fp) >> 2);

    } // end for index

fclose(fp);

// change the palette to newly loaded palette if commanded to do so

if (enable_palette)
   {

   for (index=0; index<256; index++)
       {

       Set_Palette_Register(index,(RGB_color_ptr)&image->palette[index]);

       } // end for index

   } // end if change palette

} // end PCX_Load

//////////////////////////////////////////////////////////////////////////////

void PCX_Show_Buffer(pcx_picture_ptr image)
{
// just copy he pcx buffer into the video buffer

_fmemcpy((char far *)video_buffer,
         (char far *)image->buffer,SCREEN_WIDTH*SCREEN_HEIGHT);

_redraw_screen();

} // end PCX_Show_Picture

//////////////////////////////////////////////////////////////////////////////

void Sprite_Init(sprite_ptr sprite,int x,int y,int ac,int as,int mc,int ms)
{
// this function initializes a sprite with the sent data

int index;

sprite->x            = x;
sprite->y            = y;
sprite->x_old        = x;
sprite->y_old        = y;
sprite->width        = SPRITE_WIDTH;
sprite->height       = SPRITE_HEIGHT;
sprite->anim_clock   = ac;
sprite->anim_speed   = as;
sprite->motion_clock = mc;
sprite->motion_speed = ms;
sprite->curr_frame   = 0;
sprite->state        = SPRITE_DEAD;
sprite->num_frames   = 0;
sprite->background   =  (char far *)malloc(SPRITE_WIDTH * SPRITE_HEIGHT+1);

// set all bitmap pointers to null

for (index=0; index<MAX_SPRITE_FRAMES; index++)
    sprite->frames[index] = NULL;
```

```
} // end Sprite_Init

//////////////////////////////////////////////////////////////////////////////

void Sprite_Delete(sprite_ptr sprite)
{
// this function deletes all the memory associated with a sprire

int index;

_ffree(sprite->background);

// now de-allocate all the animation frames

for (index=0; index<MAX_SPRITE_FRAMES; index++)
    _ffree(sprite->frames[index]);

} // end Sprite_Delete


//////////////////////////////////////////////////////////////////////////////

void PCX_Grap_Bitmap(pcx_picture_ptr image,
                    sprite_ptr sprite,
                    int sprite_frame,
                    int grab_x, int grab_y)

{
// this function will grap a bitmap from the pcx frame buffer. it uses the
// convention that the 320x200 pixel matrix is sub divided into a smaller
// matrix of 12x8 adjacent squares each being a 24x24 pixel bitmap
// the caller sends the pcx picture along with the sprite to save the image
// into and the frame of the sprite.  finally, the position of the bitmap
// that should be grabbed is sent

int x_off,y_off, x,y, index;
char far *sprite_data;

// first allocate the memory for the sprite in the sprite structure

sprite->frames[sprite_frame] = (char far *)malloc(SPRITE_WIDTH * SPRITE_HEIGHT);

// create an alias to the sprite frame for ease of access

sprite_data = sprite->frames[sprite_frame];

// now load the sprite data into the sprite frame array from the pcx picture

// we need to find which bitmap to scan, remember the pcx picture is really a
// 12x8 matrix of bitmaps where each bitmap is 24x24 pixels. note:0,0 is upper
// left bitmap and 11,7 is the lower right bitmap.

x_off = 25 * grab_x + 1;
y_off = 25 * grab_y + 1;

// compute starting y address

y_off = y_off * 320;

for (y=0; y<SPRITE_HEIGHT; y++)
    {

    for (x=0; x<SPRITE_WIDTH; x++)
        {

        // get the next byte of current row and place into next position in
        // sprite frame data buffer

        sprite_data[y*24 + x] = image->buffer[y_off + x_off + x];

        } // end for x
        // move to next line of picture buffer

        y_off+=320;

    } // end for y

// increment number of frames

sprite->num_frames++;

// done!, let's bail!

} // end PCX_Grap_Bitmap

//////////////////////////////////////////////////////////////////////////////

void Behind_Sprite(sprite_ptr sprite)
{

// this function scans the background behind a sprite so that when the sprite
// is draw, the background isnn'y obliterated

char far *work_back;
int work_offset=0,offset,y;

// alias a pointer to sprite background for ease of access

work_back = sprite->background;

// compute offset of background in video buffer

offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;

for (y=0; y<SPRITE_HEIGHT; y++)
    {
    // copy the next row out off screen buffer into sprite background buffer

    _fmemcpy((char far *)&work_back[work_offset],
             (char far *)&video_buffer[offset],
             SPRITE_WIDTH);

    // move to next line in video buffer and in sprite background buffer

    offset      += SCREEN_WIDTH;
    work_offset += SPRITE_WIDTH;

    } // end for y

_redraw_screen();

} // end Behind_Sprite

//////////////////////////////////////////////////////////////////////////////

void Erase_Sprite(sprite_ptr sprite)
{
// replace the background that was behind the sprite

// this function replaces the background that was saved from where a sprite
// was going to be placed

char far *work_back;
int work_offset=0,offset,y;

// alias a pointer to sprite background for ease of access

work_back = sprite->background;

// compute offset of background in video buffer
```

```
offset = (sprite->y_old << 8) + (sprite->y_old << 6) + sprite->x_old;

for (y=0; y<SPRITE_HEIGHT; y++)
    {
    // copy the next row out off screen buffer into sprite background buffer

    _fmemcpy((char far *)&video_buffer[offset],
             (char far *)&work_back[work_offset],
             SPRITE_WIDTH);

    // move to next line in video buffer and in sprite background buffer

    offset      += SCREEN_WIDTH;
    work_offset += SPRITE_WIDTH;

    } // end for y

_redraw_screen();

} // end Erase_Sprite

//////////////////////////////////////////////////////////////////////////////

void Draw_Sprite(sprite_ptr sprite)
{

// this function draws a sprite on the screen row by row very quickly
// note the use of shifting to implement multplication

char far *work_sprite;
int work_offset=0,offset,x,y;
unsigned char data;

// alias a pointer to sprite for ease of access

work_sprite = sprite->frames[sprite->curr_frame];

// compute offset of sprite in video buffer

offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;

for (y=0; y<SPRITE_HEIGHT; y++)
    {
    // copy the next row into the screen buffer using memcpy for speed

    for (x=0; x<SPRITE_WIDTH; x++)
        {

        // test for transparent pixel i.e. 0, if not transparent then draw

        if ((data=work_sprite[work_offset+x]))
             video_buffer[offset+x] = data;

        } // end for x

    // move to next line in video buffer and in sprite bitmap buffer

    offset      += SCREEN_WIDTH;
    work_offset += SPRITE_WIDTH;

    } // end for y

_redraw_screen();

} // end Draw_Sprite


// M A I N ///////////////////////////////////////////////////////////////////


void main2(void)
```

```
{

long index,redraw;
RGB_color color;
int frame_dir = 1;

pcx_picture town, cowboys;

sprite cowboy;

// set video mode to 320x200 256 color mode

Set_Mode(VGA256);
video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr
video_buffer_w= (unsigned int far *)MEMORY_0xA0000000;  // vram word ptr
rom_char_set = (unsigned char far *)MEMORY_0xF000FA6EL;

// set up the global pointers to screen ram

// Set_Screen_Pointers();

// load in background

PCX_Init((pcx_picture_ptr)&town);
PCX_Load("town.pcx", (pcx_picture_ptr)&town,1);
PCX_Show_Buffer((pcx_picture_ptr)&town);

PCX_Delete((pcx_picture_ptr)&town);

// print header

Blit_String(128, 24,50, "TOMBSTONE");

// load in the players imagery

PCX_Init((pcx_picture_ptr)&cowboys);
PCX_Load("cowboys.pcx", (pcx_picture_ptr)&cowboys,0);

// grab all the images from the cowboys pcx picture

Sprite_Init((sprite_ptr)&cowboy,SPRITE_WIDTH,100,0,7,0,3);

PCX_Grap_Bitmap((pcx_picture_ptr)&cowboys,(sprite_ptr)&cowboy,0,0,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&cowboys,(sprite_ptr)&cowboy,1,1,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&cowboys,(sprite_ptr)&cowboy,2,2,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&cowboys,(sprite_ptr)&cowboy,3,1,0);

// kill the pcx memory and buffers now that were done

PCX_Delete((pcx_picture_ptr)&cowboys);

Behind_Sprite((sprite_ptr)&cowboy);
Draw_Sprite((sprite_ptr)&cowboy);

// main loop

cowboy.state = SPRITE_ALIVE;

while(!kbhit())
     {
       FAST_CPU_WAIT(1);

     redraw = 0; // used to flag if we need a redraw

     if (cowboy.state==SPRITE_ALIVE)
        {
        // test if its time change frames

        if (++cowboy.anim_clock > cowboy.anim_speed)
           {
           // reset the animation clock
```

```
      cowboy.anim_clock = 0;

      if (++cowboy.curr_frame >= cowboy.num_frames)
         {
         cowboy.curr_frame = 0;

         } // end if reached last frame

      redraw=1;

      } // end if time to change frames

   // now test if its time to move the cowboy

   if (++cowboy.motion_clock > cowboy.motion_speed)
      {
      // reset the motion clock

      cowboy.motion_clock = 0;

      // save old position

      cowboy.x_old = cowboy.x;

      redraw = 1;

      // move cowboy

      if (++cowboy.x >= SCREEN_WIDTH-2*SPRITE_WIDTH)
         {

         Erase_Sprite((sprite_ptr)&cowboy);
         cowboy.state = SPRITE_DEAD;
         redraw         = 0;

         } // end if reached last frame

      } // end if time to change frames

   } // end if cowboy alive
else
   {
   // try and start up another cowboy

   if (rand()%100 == 0 )
      {
      cowboy.state      = SPRITE_ALIVE;
      cowboy.x          = SPRITE_WIDTH;
      cowboy.curr_frame = 0;
      cowboy.anim_speed   = 3 + rand()%6;
      cowboy.motion_speed = 1 + rand()%3;
      cowboy.anim_clock   = 0;
      cowboy.motion_clock = 0;

      Behind_Sprite((sprite_ptr)&cowboy);
      }

   } // end else dead, try to bring back to life

// now the sprite has had it's state updated

if (redraw)
   {
   // erase sprite at old position

   Erase_Sprite((sprite_ptr)&cowboy);

   // scan the background at new postition

   Behind_Sprite((sprite_ptr)&cowboy);
```

```
        // draw sprite at new position

        Draw_Sprite((sprite_ptr)&cowboy);

        // update old position

        cowboy.x_old = cowboy.x;
        cowboy.y_old = cowboy.y;

        } // end if sprites needed to be redrawn

     //Delay(1000);
     } // end while

// make a cool clear screen, disolve screen, in one line, eye might add!

for (index=0; index<=300000; index++)
{
      Plot_Pixel_Fast(rand()%320, rand()%200, 0);
      if(!(index % 200)) _redraw_screen();
};

// go back to text mode

Set_Mode(TEXT_MODE);

} // end main
```

CHAP_06

RAY.CPP

```
//--------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//--------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//--------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
//#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>
//#include <graph.h>  // we'll use microsofts stuff for this progrom

// D E F I N E S ////////////////////////////////////////////////////////////

// #define DEBUG 1

#define OVERBOARD          48 // the absolute closest a player can get to a wall

#define INTERSECTION_FOUND 1

// constants used to represent angles
```

```
#define ANGLE_0     0
#define ANGLE_1     5
#define ANGLE_2     10
#define ANGLE_4     20
#define ANGLE_5     25
#define ANGLE_6     30
#define ANGLE_15    80
#define ANGLE_30    160
#define ANGLE_45    240
#define ANGLE_60    320
#define ANGLE_90    480
#define ANGLE_135   720
#define ANGLE_180   960
#define ANGLE_225   1200
#define ANGLE_270   1440
#define ANGLE_315   1680
#define ANGLE_360   1920

#define WORLD_ROWS    16        // number of rows in the game world
#define WORLD_COLUMNS 16        // number of columns in the game world
#define CELL_X_SIZE   64        // size of a cell in the gamw world
#define CELL_Y_SIZE   64

// size of overall game world

#define WORLD_X_SIZE  (WORLD_COLUMNS * CELL_X_SIZE)
#define WORLD_Y_SIZE  (WORLD_ROWS    * CELL_Y_SIZE)

// G L O B A L S /////////////////////////////////////////////////////////////

// unsigned int far *clock = (unsigned int far *)0x0000046C; // pointer to internal
                                                          // 18.2 clicks/sec
// USAGE: ULONG now = (*clock)();
ULONG (*clock)() = PMEM_0x0000046C;


// world map of nxn cells, each cell is 64x64 pixels

char far *world[WORLD_ROWS];       // pointer to matrix of cells that make up
                                   // world

float far *tan_table;              // tangent tables used to compute initial
float far *inv_tan_table;          // intersections with ray


float far *y_step;                 // x and y steps, used to find intersections
float far *x_step;                 // after initial one is found


float far *cos_table;              // used to cacell out fishbowl effect

float far *inv_cos_table;          // used to compute distances by calculating
float far *inv_sin_table;          // the hypontenuse


// F U N C T I O N S /////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////

void Timer(int clicks)
{
//------------------------------------------
// EACH CLICK IS APPROX. 55 MILLISECONDS.
// YOU MAY USE Sleep() ... BUT IT'S NOT A PLAIN DOS FEATURE...
/*
Sleep(clicks*55);
return;
*/
//------------------------------------------
```

```
// this function uses the internal time keeper timer i.e. the one that goes
// at 18.2 clicks/sec to to a time delay.  You can find a 32 bit value of
// this timer at 0000:046Ch

unsigned int now;

// get current time

now = (*clock)();

// wait till time has gone past current time plus the amount we eanted to
// wait.  Note each click is approx. 55 milliseconds.

while(abs((*clock)() - now) < clicks){ Sleep(5); }

} // end Timer


/////////////////////////////////////////////////////////////////////////////

void Build_Tables(void)
{

int ang;
float rad_angle;

// allocate memory for all look up tables

// tangent tables equivalent to slopes

tan_table     = (float far *)_fmalloc(sizeof(float) * (ANGLE_360+1) );
inv_tan_table = (float far *)_fmalloc(sizeof(float) * (ANGLE_360+1) );

// step tables used to find next intersections, equivalent to slopes
// times width and height of cell

y_step        = (float far *)_fmalloc(sizeof(float) * (ANGLE_360+1) );
x_step        = (float far *)_fmalloc(sizeof(float) * (ANGLE_360+1) );


// cos table used to fix view distortion caused by caused by radial projection

cos_table     = (float far *)_fmalloc(sizeof(float) * (ANGLE_360+1) );


// 1/cos and 1/sin tables used to compute distance of intersection very
// quickly

inv_cos_table = (float far *)_fmalloc(sizeof(float) * (ANGLE_360+1) );
inv_sin_table = (float far *)_fmalloc(sizeof(float) * (ANGLE_360+1) );

// create tables, sit back for a sec!

for (ang=ANGLE_0; ang<=ANGLE_360; ang++)
    {

    rad_angle = (3.272e-4) + ang * 2*3.141592654/ANGLE_360;

    tan_table[ang]     = tan(rad_angle);
    inv_tan_table[ang] = 1/tan_table[ang];


    // tangent has the incorrect signs in all quadrants except 1, so
    // manually fix the signs of each quadrant since the tangent is
    // equivalent to the slope of a line and if the tangent is wrong
    // then the ray that is case will be wrong

    if (ang>=ANGLE_0 && ang<ANGLE_180)
       {
       y_step[ang]        = fabs(tan_table[ang]     * CELL_Y_SIZE);
       }
    else
       y_step[ang]        = -fabs(tan_table[ang]    * CELL_Y_SIZE);

    if (ang>=ANGLE_90 && ang<ANGLE_270)
       {
       x_step[ang]        =-fabs(inv_tan_table[ang] * CELL_X_SIZE);
       }
    else
       {
       x_step[ang]        =fabs(inv_tan_table[ang]  * CELL_X_SIZE);
       }

    // create the sin and cosine tables to copute distances

    inv_cos_table[ang] = 1/cos(rad_angle);
    inv_sin_table[ang] = 1/sin(rad_angle);

    } // end for ang

// create view filter table.  There is a cosine wave modulated on top of
// the view distance as a side effect of casting from a fixed point.
// to cancell this effect out, we multiple by the inverse of the cosine
// and the result is the proper scale.  Without this we would see a
// fishbowl effect, which might be desired in some cases?

for (ang=-ANGLE_30; ang<=ANGLE_30; ang++)
    {

    rad_angle = (3.272e-4) + ang * 2*3.141592654/ANGLE_360;

    cos_table[ang+ANGLE_30] = 1/cos(rad_angle);

    } // end for

} // end Build_Tables

/////////////////////////////////////////////////////////////////////////////

void Allocate_World(void)
{
// this function allocates the memory for the world

int index;

// allocate each row

for (index=0; index<WORLD_ROWS; index++)
    {
    world[index] = (char far *)_fmalloc(WORLD_COLUMNS+1);

    } // end for index

} // end Allocate_World


////////////////////////////////////////////////////////////////////////////////

int Load_World(char *file)
{
// this function opens the input file and loads the world data from it

FILE  *fp /*, *fopen() */;
int index,row,column;
char buffer[WORLD_COLUMNS+2],ch;

// open the file

if (!(fp = fopen(file,"r")))
   return(0);
```

```
// load in the data

for (row=0; row<WORLD_ROWS; row++)
    {
    // load in the next row

    for (column=0; column<WORLD_COLUMNS; column++)
        {

        while((ch = getc(fp))==10){} // filter out CR

        // translate character to integer

        if (ch == ' ')
           ch=0;
        else
           ch = ch - '0';

        // insert data into world

        world[row][column] = ch;

        } // end for column

    // process the row

    } // end for row

// close the file

fclose(fp);

return(1);

} // end Load_World

/////////////////////////////////////////////////////////////////////////////

void sline(long x1, long y1,long x2, long y2,  int color)
{

// used a a diagnostic function to draw a scaled line

x1 = x1 / 4;
y1 = 256 - ( y1 / 4);

x2 = x2 / 4;
y2 = 256 - ( y2 / 4);

_setcolor(color);
_moveto((int)x1,(int)y1);
_lineto((int)x2,(int)y2);

} // end sline

/////////////////////////////////////////////////////////////////////////////

void splot(long x,long y,int color)
{
// used as a diagnostic function to draw a scaled point

x = x / 4;
y = 256 - ( y / 4);

_setcolor(color);

_setpixel((int)x,    (int)y);
_setpixel((int)x+1, (int)y);
_setpixel((int)x,    (int)y+1);
_setpixel((int)x+1, (int)y+1);
```

```
} // end splot

//////////////////////////////////////////////////////////////////////////////

void Draw_2D_Map(void)
{
// draw 2-D map of world

int row,column,block,t,done=0;

for (row=0; row<WORLD_ROWS; row++)
   {
   for (column=0; column<WORLD_COLUMNS; column++)
       {

       block = world[row][column];

       // test if there is a solid block there

       if (block==0)
          {

          _setcolor(15);
          _rectangle(_GBORDER,column*CELL_X_SIZE/4,row*CELL_Y_SIZE/4,
                        column*CELL_X_SIZE/4+CELL_X_SIZE/4-
1,row*CELL_Y_SIZE/4+CELL_Y_SIZE/4-1);

          }
       else
          {

          _setcolor(2);
          _rectangle(_GFILLINTERIOR,column*CELL_X_SIZE/4,row*CELL_Y_SIZE/4,
                        column*CELL_X_SIZE/4+CELL_X_SIZE/4-
1,row*CELL_Y_SIZE/4+CELL_Y_SIZE/4-1);

          }

       } // end for column

   } // end for row

} // end Draw_2D_Map

//////////////////////////////////////////////////////////////////////////////

void Ray_Caster(long x,long y,long view_angle)
{
// This function casts out 320 rays from the viewer and builds up the video
// display based on the intersections with the walls. The 320 rays are
// cast in such a way that they all fit into a 60 degree field of view
// a ray is cast and then the distance to the first horizontal and vertical
// edge that has a cell in it is recorded.  The intersection that has the
// closer distance to the user is the one that is used to draw the bitmap.
// the distance is used to compute the height of the "sliver" of texture
// or line that will be drawn on the screen

// note: this function uses floating point (slow), no optimizations (slower)
// and finally it makes calls to Microsofts Graphics libraries (slowest!)
// however, writing it in this manner makes it many orders of magnitude
// easier to understand.

int rcolor;

long xray=0,         // tracks the progress of a ray looking for Y interesctions
     yray=0,         // tracks the progress of a ray looking for X interesctions
     next_y_cell,    // used to figure out the quadrant of the ray
     next_x_cell,
     cell_x,         // the current cell that the ray is in
     cell_y,
     x_bound,        // the next vertical and horizontal intersection point
     y_bound,
     xb_save,       // storage to record intersections cell boundaries
     yb_save,
     x_delta,       // the amount needed to move to get to the next cell
     y_delta,       // position
     ray,           // the current ray being cast 0-320
     casting=2,     // tracks the progress of the X and Y component of the ray
     x_hit_type,    // records the block that was intersected, used to figure
     y_hit_type,    // out which texture to use

     top,           // used to compute the top and bottom of the sliver that
     bottom;        // is drawn symetrically around the bisecting plane of the
                    // screens vertical extents


float xi,           // used to track the x and y intersections
      yi,
      xi_save,      // used to save exact x and y intersection points
      yi_save,
      dist_x,       // the distance of the x and y ray intersections from
      dist_y,       // the viewpoint
      scale;        // the final scale to draw the "sliver" in

// S E C T I O N  1 ////////////////////////////////////////////////////////v

// initialization

// compute starting angle from player.  Field of view is 60 degrees, so
// subtract half of that current view angle

if ( (view_angle-=ANGLE_30) < 0)
   {
   // wrap angle around
   view_angle=ANGLE_360 + view_angle;
   } // end if

rcolor=1 + rand()%14;

// loop through all 320 rays

// section 2

for (ray=0; ray<320; ray++)
    {

// S E C T I O N  2 /////////////////////////////////////////////////////////

    // compute first x intersection

    // need to know which half plane we are casting from relative to Y axis

    if (view_angle >= ANGLE_0 && view_angle < ANGLE_180)
       {

       // compute first horizontal line that could be intersected with ray
       // note: it will be above player

       y_bound = CELL_Y_SIZE + CELL_Y_SIZE * (y / CELL_Y_SIZE);

       // compute delta to get to next horizontal line

       y_delta = CELL_Y_SIZE;

       // based on first possible horizontal intersection line, compute X
       // intercept, so that casting can begin

       xi = inv_tan_table[view_angle] * (y_bound - y) + x;

       // set cell delta

       next_y_cell = 0;
```

```
         } // end if upper half plane
      else
         {

         // compute first horizontal line that could be intersected with ray
         // note: it will be below player

         y_bound = CELL_Y_SIZE * (y / CELL_Y_SIZE);

         // compute delta to get to next horizontal line

         y_delta = -CELL_Y_SIZE;

         // based on first possible horizontal intersection line, compute X
         // intercept, so that casting can begin

         xi = inv_tan_table[view_angle] * (y_bound - y) + x;

         // set cell delta

         next_y_cell = -1;

         } // end else lower half plane


// S E C T I O N  3 ////////////////////////////////////////////////////////

      // compute first y intersection

      // need to know which half plane we are casting from relative to X axis

      if (view_angle < ANGLE_90 || view_angle >= ANGLE_270)
         {

         // compute first vertical line that could be intersected with ray
         // note: it will be to the right of player

         x_bound = CELL_X_SIZE + CELL_X_SIZE * (x / CELL_X_SIZE);

         // compute delta to get to next vertical line

         x_delta = CELL_X_SIZE;

         // based on first possible vertical intersection line, compute Y
         // intercept, so that casting can begin

         yi = tan_table[view_angle] * (x_bound - x) + y;

         // set cell delta

         next_x_cell = 0;

         } // end if right half plane
      else
         {

         // compute first vertical line that could be intersected with ray
         // note: it will be to the left of player

         x_bound = CELL_X_SIZE * (x / CELL_X_SIZE);

         // compute delta to get to next vertical line

         x_delta = -CELL_X_SIZE;

         // based on first possible vertical intersection line, compute Y
         // intercept, so that casting can begin

         yi = tan_table[view_angle] * (x_bound - x) + y;
```

```
       // set cell delta

       next_x_cell = -1;

       } // end else right half plane

// begin cast

    casting       = 2;                 // two rays to cast simultaneously
    xray=yray     = 0;                 // reset intersection flags


// S E C T I O N  4 /////////////////////////////////////////////////////////

    while(casting)
         {

         // continue casting each ray in parallel

         if (xray!=INTERSECTION_FOUND)
            {

            // test for asymtotic ray

            // if (view_angle==ANGLE_90 || view_angle==ANGLE_270)

            if (fabs(y_step[view_angle])==0)
               {
               xray = INTERSECTION_FOUND;
               casting--;
               dist_x = 1e+8;

               } // end if asymtotic ray

            // compute current map position to inspect

            cell_x = ( (x_bound+next_x_cell) / CELL_X_SIZE);
            cell_y = (long)(yi / CELL_Y_SIZE);

                    //------------------------------------------
                    // ALGORITHM ERRORS FIXING
                    //------------------------------------------
                    if(cell_y > (WORLD_ROWS-1)) cell_y = (WORLD_ROWS-1);
                    if(cell_y < 0) cell_y = 0;
                    if(cell_x > (WORLD_COLUMNS-1)) cell_x = (WORLD_COLUMNS-1);
                    if(cell_x < 0) cell_x = 0;
                    //------------------------------------------

            // test if there is a block where the current x ray is intersecting

            if ((x_hit_type = world[(WORLD_ROWS-1) - cell_y][cell_x])!=0)
               {
               // compute distance

               dist_x  = (yi - y) * inv_sin_table[view_angle];
               yi_save = yi;
               xb_save = x_bound;

               // terminate X casting

               xray = INTERSECTION_FOUND;
               casting--;

               } // end if a hit
            else
               {
               // compute next Y intercept

               yi += y_step[view_angle];

               } // end else
            } // end if x ray has intersected

// S E C T I O N  5 /////////////////////////////////////////////////////////

         if (yray!=INTERSECTION_FOUND)
            {

            // test for asymtotic ray

            // if (view_angle==ANGLE_0 || view_angle==ANGLE_180)

            if (fabs(x_step[view_angle])==0)
               {
               yray = INTERSECTION_FOUND;
               casting--;
               dist_y=1e+8;

               } // end if asymtotic ray

            // compute current map position to inspect

            cell_x = (long)(xi / CELL_X_SIZE);
            cell_y = ( (y_bound + next_y_cell) / CELL_Y_SIZE);

                    //------------------------------------------
                    // ALGORITHM ERRORS FIXING
                    //------------------------------------------
                    if(cell_y > (WORLD_ROWS-1)) cell_y = (WORLD_ROWS-1);
                    if(cell_y < 0) cell_y = 0;
                    if(cell_x > (WORLD_COLUMNS-1)) cell_x = (WORLD_COLUMNS-1);
                    if(cell_x < 0) cell_x = 0;
                    //------------------------------------------


            // test if there is a block where the current y ray is intersecting

            if ((y_hit_type = world[(WORLD_ROWS-1) - cell_y][cell_x])!=0)
               {
               // compute distance

               dist_y  = (xi - x) * inv_cos_table[view_angle];
               xi_save = xi;
               yb_save = y_bound;

               // terminate Y casting

               yray = INTERSECTION_FOUND;
               casting--;

               } // end if a hit
            else
               {
               // compute next X intercept

               xi += x_step[view_angle];

               } // end else

            } // end if y ray has intersected

         // move to next possible intersection points


         x_bound += x_delta;
         y_bound += y_delta;


         // _settextposition(38,40);
         // printf("x_bound = %ld, y_bound = %ld    ",x_bound,y_bound);
         } // end while not done

// S E C T I O N  6 ////////////////////////////////////////////////////////

    // at this point, we know that the ray has succesfully hit both a
    // vertical wall and a horizontal wall, so we need to see which one
    // was closer and then render it

    // note: latter we will replace the crude monochrome line with a sliver
    // of texture, but this is good enough for now

    if (dist_x < dist_y)
       {

       sline(x,y,(long)xb_save,(long)yi_save,rcolor);

       // there was a vertical wall closer than the horizontal

       // compute actual scale and multiply by view filter so that spherical
       // distortion is cancelled

       scale = cos_table[ray]*15000/(1e-10 + dist_x);

       // compute top and bottom and do a very crude clip

       if ( (top    = 100 - scale/2) < 1)
          top = 1;

       if ( (bottom = top+scale) > 200)
          bottom=200;

       // draw wall sliver and place some dividers up

       if ( ((long)yi_save) % CELL_Y_SIZE <= 1 )
          _setcolor(15);
       else
          _setcolor(10);

       _moveto((int)(638-ray),(int)top);
       _lineto((int)(638-ray),(int)bottom);

       }
    else // must of hit a horizontal wall first
       {

       sline(x,y,(long)xi_save,(long)yb_save,rcolor);

       // compute actual scale and multiply by view filter so that spherical
       // distortion is cancelled

       scale = cos_table[ray]*15000/(1e-10 + dist_y);

       // compute top and bottom and do a very crude clip

       if ( (top    = 100 - scale/2) < 1)
          top = 1;

       if ( (bottom = top+scale) > 200)
          bottom=200;

       // draw wall sliver and place some dividers up

       if ( ((long)xi_save) % CELL_X_SIZE <= 1 )
          _setcolor(15);
       else
          _setcolor(2);


       _moveto((int)(638-ray),(int)top);
       _lineto((int)(638-ray),(int)bottom);
       } // end else

// S E C T I O N  7 //////////////////////////////////////////////////////////

    // cast next ray

    if (++view_angle>=ANGLE_360)
       {
       // reset angle back to zero

       view_angle=0;

       } // end if

    } // end for ray

} // end Ray_Caster

// M A I N ///////////////////////////////////////////////////////////////////

void main2(void)
{

int row,column,block,t,done=0;

long x,y,view_angle,x_cell,y_cell,x_sub_cell,y_sub_cell;

float dx,dy;

// seed random number genrerator

srand(13);

// set mode to 640x480 so we can fit a lot of info on the screen for
// educational purposes

_setvideomode(_VRES16COLOR);
_set_render_options(TRUE, RENDER_MANUAL_REDRAW);

Allocate_World();

// build all the lookuo tables

Build_Tables();

Load_World("raymap.dat");

// draw top view of world

Draw_2D_Map();
_redraw_screen();

// draw information prompts

_settextposition(18,8);

printf("2-D Map View");

_settextposition(16,54);

printf("3-D Projection");

_settextposition(35,16);

printf("Use numeric keypad to move. Press Q to quit.");

// draw window around view port

_setcolor(15);
_rectangle(_GBORDER,318,0,639,201);
```

```
_redraw_screen();

x=8*64+25;
y=3*64+25;
view_angle=ANGLE_60;

// render initial view

Ray_Caster(x,y,view_angle);
_redraw_screen();

// wait for user to press q to quit

while(!done)
     {
      FAST_CPU_WAIT(10);

     // has keyboard been hit?

     if (kbhit())
        {

        // reset deltas

        dx=dy=0;

        // clear viewport

        _setcolor(0);
        _rectangle(_GFILLINTERIOR,319,1,638,200);
        _setcolor(8);
        _rectangle(_GFILLINTERIOR,319,100,638,200);
_redraw_screen();

        // what is user doing

        switch(getch())
               {
               case '6':
                       {
                       if ((view_angle-=ANGLE_6)<ANGLE_0)
                          view_angle=ANGLE_360;

                       } break;
               case '4':
                       {
                       if ((view_angle+=ANGLE_6)>=ANGLE_360)
                          view_angle=ANGLE_0;

                       } break;


               case '8':
                       {

                       // move player along view vector foward

                       dx=cos(6.28*view_angle/ANGLE_360)*10;
                       dy=sin(6.28*view_angle/ANGLE_360)*10;

                       } break;

               case '2':
                       {
                       // move player along view vector backward

                       dx=-cos(6.28*view_angle/ANGLE_360)*10;
                       dy=-sin(6.28*view_angle/ANGLE_360)*10;

                        // test if player is bumping into a wall
                       } break;

               case 'q': { done=1; } break;

              default:break;

               } // end switch

        // move player

        x+=dx;
        y+=dy;

        // test if user has bumped into a wall i.e. test if there
        // is a cell within the direction of motion, if so back up !

        // compute cell position

        x_cell = x/CELL_X_SIZE;
        y_cell = y/CELL_Y_SIZE;

        // compute position relative to cell

        x_sub_cell = x % CELL_X_SIZE;
        y_sub_cell = y % CELL_Y_SIZE;


        // resolve motion into it's x and y components

        if (dx>0 )
           {
           // moving right

           if ( (world[(WORLD_ROWS-1) - y_cell][x_cell+1] != 0)  &&
                (x_sub_cell > (CELL_X_SIZE-OVERBOARD ) ) )
                {
                // back player up amount he steped over the line

                x-= (x_sub_cell-(CELL_X_SIZE-OVERBOARD ));

                } // end if need to back up

           }
        else
           {
           // moving left

           if ( (world[(WORLD_ROWS-1) - y_cell][x_cell-1] != 0)  &&
                (x_sub_cell < (OVERBOARD) ) )
                {
                // back player up amount he steped over the line

                x+= (OVERBOARD-x_sub_cell) ;

                } // end if need to back up

           } // end else

        if (dy>0 )
           {
           // moving up

           if ( (world[(WORLD_ROWS-1) - (y_cell+1)][x_cell] != 0)  &&
                (y_sub_cell > (CELL_Y_SIZE-OVERBOARD ) ) )
                {
                // back player up amount he steped over the line

                y-= (y_sub_cell-(CELL_Y_SIZE-OVERBOARD ));
```

```
                } // end if need to back up
           }
        else
           {
           // moving down

           if ( (world[(WORLD_ROWS-1) - (y_cell-1)][x_cell] != 0)  &&
                (y_sub_cell < (OVERBOARD) ) )
                {
                // back player up amount he steped over the line

                y+= (OVERBOARD-y_sub_cell);

                } // end if need to back up

           } // end else

        // render the view

        Ray_Caster(x,y,view_angle);
             _redraw_screen();

        // display status

        _settextposition(20,1);

        printf("\nPosition of player is (%ld,%ld)   ",x,y);
        printf("\nView angle is %ld  ",(long)(360*(float)view_angle/ANGLE_360));
        printf("\nCurrent cell is (%ld,%ld)   ",x_cell,y_cell);
        printf("\nRelative position within cell is (%ld,%ld)  ",
                x_sub_cell,y_sub_cell);

        }  // end if kbhit

     } //end while

// restore original mode

_setvideomode(_DEFAULTMODE);

} // end main
```

CHAP_07

CIRCLES.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <stdio.h>
#include <math.h>
// #include <graph.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>

// D E F I N E S ////////////////////////////////////////////////////////////
```

```
#define SCREEN_WIDTH      (unsigned int)320
#define SCREEN_HEIGHT     (unsigned int)200

// G L O B A L S ////////////////////////////////////////////////////////////

unsigned char far *video_buffer = (unsigned char far *)0xA0000000L; // vram byte ptr
unsigned char far *double_buffer = NULL;

// F U N C T I O N S ////////////////////////////////////////////////////////

void Init_Double_Buffer(void)
{

double_buffer = (unsigned char far *)_fmalloc(SCREEN_WIDTH * SCREEN_HEIGHT + 1);

_fmemset(double_buffer, 0, SCREEN_WIDTH * SCREEN_HEIGHT + 1);

} // end Init_Double_Buffer

/////////////////////////////////////////////////////////////////////////////

void Show_Double_Buffer(char far *buffer)
{
// copy the double buffer into the video buffer

      memcpy(video_buffer, buffer, 320*200);
      _redraw_screen();
      return;

//-------------------------------------------
// NO ANY ASM
//-------------------------------------------
/*
_asm
   {
   push ds                  // save the data segment
   les di, video_buffer     // set destination i.e. video buffer
   lds si, buffer           // set source i.e. double buffer
   mov cx,320*200/2         // want to move 320*200 bytes or half that # of
   cld                      // words.
   rep movsw                // do the movement
   pop ds                   // restore the data segment
   }
*/
//-------------------------------------------

} // end Show_Double_Buffer

/////////////////////////////////////////////////////////////////////////////

void Plot_Pixel_Fast_D(int x,int y,unsigned char color)
{

// plots pixels into the double buffer

// use the fact that 320*y = 256*y + 64*y = y<<8 + y<<6

double_buffer[((y<<8) + (y<<6)) + x] = color;

} // end Plot_Pixel_Fast_D

/////////////////////////////////////////////////////////////////////////////

void Circles(void)
{
// this function draw 1000 circles into the double buffer, in a game we would
// never use a crude algorithm, like this to draw circles, we would use
// look up tables or other means; however, we just want something to be drawn
// in the double buffer
int index,xo,yo,radius,x,y,color,ang;

// draw 100 circles at random positions with random colors and sizes

for (index=0; index<1000; index++)
    {

    // get parameters for next circle

    xo     = 20 + rand()%300;
    yo     = 20 + rand()%180;
    radius = 1 + rand()%20;
    color  = rand()%256;

    for (ang=0; ang<360; ang++)
        {

        x = xo + cos(ang*3.14/180) * radius;
        y = yo + sin(ang*3.14/180) * radius;

        Plot_Pixel_Fast_D(x,y,(unsigned char)color);

        } // end ang

    } // end index

} // end Circles


// M A I N /////////////////////////////////////////////////////////////

void main2(void)
{

// set the videomode to 320x256x256

_setvideomode(_MRES256COLOR);
video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr

// create a double buffer and clear it

Init_Double_Buffer();

_settextposition(0,0);
printf("Drawing 1000 circles to double buffer. \nPlease wait...");

// draw the circles to the double buffer

Circles();

printf("Done, press any key.");

// wait for user to hit key then blast double buffer to video screen

getch();

Show_Double_Buffer((char*)double_buffer);

_settextposition(0,0);
printf("That was quick. Hit any key to exit.");

getch();

// restore video mode

_setvideomode(_DEFAULTMODE);

} // end main
```

VSYNC.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <dos.h>
//#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
//#include <graph.h>

// D E F I N E S ////////////////////////////////////////////////////////////

#define VGA_INPUT_STATUS_1   0x3DA // vga status reg 1, bit 3 is the vsync
                                   // when 1 - retrace in progress
                                   // when 0 - no retrace

#define VGA_VSYNC_MASK 0x08        // masks off unwanted bit of status reg


// G L O B A L S ////////////////////////////////////////////////////////////

unsigned char far *video_buffer = (unsigned char far *)0xA0000000L; // vram byte ptr


// F U N C T I O N S ////////////////////////////////////////////////////////

void Wait_For_Vsync(void )
{
// this function waits for the start of a vertical retrace, if a vertical
// retrace is in progress then it waits until the next one

while(_inp(VGA_INPUT_STATUS_1) & VGA_VSYNC_MASK)
     {
     // do nothing, vga is in retrace
     } // end while

// now wait for vysnc and exit

while(!(_inp(VGA_INPUT_STATUS_1) & VGA_VSYNC_MASK))
     {
     // do nothing, wait for start of retrace
     } // end while

// at this point a vertical retrace is occuring, so return back to caller

} // Wait_For_Vsync

// M A I N /////////////////////////////////////////////////////////////////

void main2(void )
{

long number_vsyncs=0;  // tracks number of retrace cycles

while(!kbhit())
     {

     // wait for a vsync

     Wait_For_Vsync();
     // do graphics or whatever now that we know electron gun is retracing
     // we only have 1/70 of a second though! Usually, we would copy the
     // double buffer to the video ram

     // ....

     // tally vsyncs

     number_vsyncs++;

     // print to screen

     _settextposition(0,0);
     printf("Number of vsync's = %ld   ",number_vsyncs);


     } // end while

} // end main
```

BIRDANI.CPP

```
//-------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//-------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//-------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
// #include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>

#include "graph0.h" // include our graphics library

// D E F I N E S /////////////////////////////////////////////////////////////

#define BIRD_START_COLOR_REG 16
#define BIRD_END_COLOR_REG   28

// G L O B A L S /////////////////////////////////////////////////////////////

// unsigned int far *clock = (unsigned int far *)0x0000046C; // pointer to internal
                                                          // 18.2 clicks/sec
// USAGE: ULONG now = (*clock)();
ULONG (*clock)() = PMEM_0x0000046C;

pcx_picture birds;

//////////////////////////////////////////////////////////////////////////////

void Timer(int clicks)
{
//-------------------------------------------
// EACH CLICK IS APPROX. 55 MILLISECONDS.
// YOU MAY USE Sleep() ... BUT IT'S NOT A PLAIN DOS FEATURE...
/*
Sleep(clicks*55);
return;
*/
//------------------------------------------


// this function uses the internal time keeper timer i.e. the one that goes
// at 18.2 clicks/sec to to a time delay.  You can find a 32 bit value of
// this timer at 0000:046Ch

unsigned int now;

// get current time

now = (*clock)();

// wait till time has gone past current time plus the amount we eanted to
// wait.  Note each click is approx. 55 milliseconds.

while(abs((*clock)() - now) < clicks){ Sleep(5); }

} // end Timer


//////////////////////////////////////////////////////////////////////////////

void Animate_Birds(void)
{
// this function animates a bird drawn with 13 different colors by turning
// on a single color and turning off all the others in a sequence

RGB_color color_1, color_2;
int index;

// clear out each of the color registers used by birds

color_1.red   = 0;
color_1.green = 0;
color_1.blue  = 0;

color_2.red   = 0;
color_2.green = 63;
color_2.blue  = 0;

// clear all the colors out

for (index=BIRD_START_COLOR_REG; index<=BIRD_END_COLOR_REG; index++)
    {

    Set_Palette_Register(index, (RGB_color_ptr)&color_1);

    } // end for index

// make first bird green and then rotate colors

Set_Palette_Register(BIRD_START_COLOR_REG, (RGB_color_ptr)&color_2);

// animate the colors

while(!kbhit())
     {
     // rotate colors

     Get_Palette_Register(BIRD_END_COLOR_REG,(RGB_color_ptr)&color_1);

     for (index=BIRD_END_COLOR_REG-1; index>=BIRD_START_COLOR_REG; index--)
         {

         Get_Palette_Register(index,(RGB_color_ptr)&color_2);
         Set_Palette_Register(index+1,(RGB_color_ptr)&color_2);

         } // end for

         Set_Palette_Register(BIRD_START_COLOR_REG,(RGB_color_ptr)&color_1);

     // wait a while

     Timer(3);

     } // end while

} // end Animate_Birds

// M A I N ///////////////////////////////////////////////////////////////////

void main2(void)
{
int index,
    done=0;

// set video mode to 320x200 256 color mode

Set_Mode(VGA256);
video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr
rom_char_set = (unsigned char far *)VGA_FONT_8X8;

// initialize the pcx file that holds all the birds

PCX_Init((pcx_picture_ptr)&birds);

// load the pcx file that holds the cells

PCX_Load("birds.pcx", (pcx_picture_ptr)&birds,1);

PCX_Show_Buffer((pcx_picture_ptr)&birds);

PCX_Delete((pcx_picture_ptr)&birds);

_settextposition(0,0);
printf("Hit any key to see animation.");

getch();

_settextposition(0,0);
printf("Hit any key to Exit.           ");

Animate_Birds();

// go back to text mode

Set_Mode(TEXT_MODE);

} // end main
```

STICK.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------

// I N C L U D E S ///////////////////////////////////////////////////////////

#include <io.h>
```

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
//#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>

#include "graph0.h" // include our graphics library

// D E F I N E S ////////////////////////////////////////////////////////////

#define VEL_CONST -1  // flags that motion should use constant velocity

// G L O B A L S ////////////////////////////////////////////////////////////

// unsigned int far *clock = (unsigned int far *)0x0000046C; // pointer to internal
                                                            // 18.2 clicks/sec
// USAGE: ULONG now = (*clock)();
ULONG (*clock)() = PMEM_0x0000046C;

sprite object;
pcx_picture stick_cells,
            street_cells;


// motion lookup table, has a separte entry for each frame of animation so
// a more realistic movement can be made based on the current frame

int object_vel[] = {17,0,6,2,3,0,17,0,6,2,3,0};


////////////////////////////////////////////////////////////////////////////////

// F U N C T I O N S ////////////////////////////////////////////////////////

void Timer(int clicks)
{
//-----------------------------------------
// EACH CLICK IS APPROX. 55 MILLISECONDS.
// YOU MAY USE Sleep() ... BUT IT'S NOT A PLAIN DOS FEATURE...
/*
Sleep(clicks*55);
return;
*/
//-----------------------------------------


// this function uses the internal time keeper timer i.e. the one that goes
// at 18.2 clicks/sec to to a time delay.  You can find a 32 bit value of
// this timer at 0000:046Ch

unsigned int now;

// get current time

now = (*clock)();

// wait till time has gone past current time plus the amount we eanted to
// wait.  Note each click is approx. 55 milliseconds.

while(abs((*clock)() - now) < clicks){ Sleep(5); }

} // end Timer


// M A I N //////////////////////////////////////////////////////////////////
```

```
void main2(void )
{
int index,
    done=0,
    vel_state=VEL_CONST;

// set video mode to 320x200 256 color mode

Set_Mode(VGA256);
video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr
rom_char_set = (unsigned char far *)VGA_FONT_8X8;

// set sprite system size so that functions use correct sprite size

sprite_width = 32;
sprite_height = 64;

// initialize the pcx file that holds the street

PCX_Init((pcx_picture_ptr)&street_cells);

// load the pcx file that holds the cells

PCX_Load("street.pcx", (pcx_picture_ptr)&street_cells,1);

PCX_Show_Buffer((pcx_picture_ptr)&street_cells);

// use the pcx buffer for the double buffer

double_buffer = (unsigned char *)street_cells.buffer;

Sprite_Init((sprite_ptr)&object,0,0,0,0,0,0);

// initialize the pcx file that holds the stickman

PCX_Init((pcx_picture_ptr)&stick_cells);

// load the pcx file that holds the cells

PCX_Load("stickman.pcx", (pcx_picture_ptr)&stick_cells,1);

// grap 6 walking frames

PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,(sprite_ptr)&object,0,0,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,(sprite_ptr)&object,1,1,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,(sprite_ptr)&object,2,2,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,(sprite_ptr)&object,3,3,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,(sprite_ptr)&object,4,4,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,(sprite_ptr)&object,5,5,0);

PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,(sprite_ptr)&object,6, 0,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,(sprite_ptr)&object,7, 1,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,(sprite_ptr)&object,8, 2,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,(sprite_ptr)&object,9, 3,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,(sprite_ptr)&object,10,4,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&stick_cells,(sprite_ptr)&object,11,5,1);



// dont need the stickman pcx file anymore

PCX_Delete((pcx_picture_ptr)&stick_cells);


// set up stickman

object.x          = 10;
object.y          = 120;
object.curr_frame = 0;

// scan background
```

```
Behind_Sprite((sprite_ptr)&object);

// main loop

while(!done)
     {

     // erase sprite

     Erase_Sprite((sprite_ptr)&object);

     // increment current frame of stickman

     if  (++object.curr_frame > 11)
          object.curr_frame = 0;

     // move sprite using constant velocity or lookup table

     if (vel_state==VEL_CONST)
        {
        object.x+=4;
        } // end if constant velocoty mode
     else
        {
        // use current frame to index into table

        object.x += object_vel[object.curr_frame];

        } // end else use lookup table to a more realistic motion

     // test if stickman is off screen

     if (object.x > 280)
          object.x=10;

     // scan background

     Behind_Sprite((sprite_ptr)&object);

     // draw sprite

     Draw_Sprite((sprite_ptr)&object);

     // copy double buffer to screen

     Show_Double_Buffer((char *)double_buffer);

     // wait a bit

     Timer(2);

     // test if user is hitting keyboard

     if (kbhit())
        {
        switch(getch())
              {
              case ' ': // toggle motion mode
                      {
                      vel_state = -vel_state;
                      } break;

              case 'q': // exit system
                      {
                      done=1;

                      } break;

              } // end switch
        } // end if kbhit

     } // end while

// delete the pcx file

PCX_Delete((pcx_picture_ptr)&street_cells);

// go back to text mode

Set_Mode(TEXT_MODE);

} // end main
```

DEFEND.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <stdio.h>
#include <math.h>
// #include <graph.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>

// D E F I N E S /////////////////////////////////////////////////////////////

#define SCREEN_WIDTH      (unsigned int)320
#define SCREEN_HEIGHT     (unsigned int)200

// G L O B A L S  ////////////////////////////////////////////////////////////

unsigned char far *video_buffer = (unsigned char far *)0xA0000000L; // vram byte ptr
unsigned char far *double_buffer = NULL;

// F U N C T I O N S /////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////

void Show_View_Port(char far *buffer,int pos)
{
// copy a portion of the double buffer to the video screen

unsigned int y,double_off, screen_off;

// there are 100 rows that need to be moved, move the data row by row

for (y=0; y<100; y++)
    {

    // compute starting offset into double buffer
    // y * 640 + pos

    double_off = ((y<<9) + (y<<7) + pos );

    // compute starting offset in video ram
    // y * 320 + 80

    screen_off = (((y+50)<<8) + ((y+50)<<6) + 80 );
    // move the data

    _fmemmove((char far *)&video_buffer[screen_off],
              (char far *)&double_buffer[double_off],160);

    } // end for y

_redraw_screen();

} // end Show_View_Port

//////////////////////////////////////////////////////////////////////////////

void Plot_Pixel_Fast_D2(int x,int y,unsigned char color)
{

// plots pixels into the double buffer with our new virtual screen size
// of 640x100

// use the fact that 640*y = 512*y + 128*y = y<<9 + y<<7

double_buffer[((y<<9) + (y<<7)) + x] = color;

} // end Plot_Pixel_Fast_D2

/////////////////////////////////////////////////////////////////////////////

void Draw_Terrain(void)
{

// this function draws the terrain into the double buffer, which in this case
// is thought of as being 640x100 pixels

int x,y=70,index;


// clear out memory first

_fmemset(double_buffer,0,(unsigned int)640*(unsigned int)100);

// draw a few stars

for (index=0; index<200; index++)
    {
    Plot_Pixel_Fast_D2(rand()%640,rand()%70,15);
    } // end for index

// draw some moutains

for (x=0; x<640; x++)
    {

    // compute offset

    y+=-1 + rand()%3;

    // make sure terrain stays within resonable boundary

    if (y>90) y=90;
    else
    if (y<40) y=40;

    // plot the dot in the double buffer

    Plot_Pixel_Fast_D2(x,y,10);

    } // end for x

} // end Draw_Terrain
```

```
// M A I N //////////////////////////////////////////////////////////////

void main2(void)
{

int done=0,sx=0;

// set the videomode to 320x256x256

_setvideomode(_MRES256COLOR);
video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr

_settextposition(0,0);

printf("Use < > to move. Press Q to quit.");

// draw a little window

_setcolor(1);

_rectangle(_GBORDER, 80-1,50-1,240+1,150+1);

// allocate memory for double buffer

double_buffer = (unsigned char *)_fmalloc(SCREEN_WIDTH * SCREEN_HEIGHT+1);

Draw_Terrain();

Show_View_Port((char *)double_buffer,sx);

// main loop

while(!done)
     {
       FAST_CPU_WAIT(10);

     // has user hit a key

     if (kbhit())
        {

        switch(getch())
              {
              case ',': // move window to left, but don't go too far
                      {
                      sx-=2;

                      if (sx<0)
                          sx=0;

                      } break;

              case '.': // move window to right, but dont go too far
                      {
                      sx+=2;

                      if (sx > 640-160)
                         sx=640-160;

                      } break;

              case 'q': // user trying to bail ?
                      {
                      done=1;

                      } break;

              } // end switch

        // copy view port to screen
        Show_View_Port((char *)double_buffer,sx);

        _settextposition(24,0);

        printf("Viewport position = %d  ",sx);

        } // end if

     } // end while

// restore video mode

_setvideomode(_DEFAULTMODE);

} // end main
```

SCREENFX.CPP

```
//-------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//-------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//-------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
//#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>

#include "graph0.h" // include our graphics library

// D E F I N E S ////////////////////////////////////////////////////////////


// S T R U C T U R E S //////////////////////////////////////////////////////

typedef struct worm_typ
        {
        int y;       // current y position of worm
        int color;   // color of worm
        int speed;   // speed of worm
        int counter; // counter

        } worm, *worm_ptr;

// G L O B A L S ////////////////////////////////////////////////////////////

// unsigned int far *clock = (unsigned int far *)0x0000046C; // pointer to internal
                                                           // 18.2 clicks/sec
// USAGE: ULONG now = (*clock)();
ULONG (*clock)() = PMEM_0x0000046C;

pcx_picture screen_fx; // our test screen

worm worms[320]; // used to make the screen melt

////////////////////////////////////////////////////////////////////////////
```

```
void Timer(int clicks)
{
//-----------------------------------------
// EACH CLICK IS APPROX. 55 MILLISECONDS.
// YOU MAY USE Sleep() ... BUT IT'S NOT A PLAIN DOS FEATURE...
/*
Sleep(clicks*55);
return;
*/
//-----------------------------------------


// this function uses the internal time keeper timer i.e. the one that goes
// at 18.2 clicks/sec to to a time delay.  You can find a 32 bit value of
// this timer at 0000:046Ch

unsigned int now;

// get current time

now = (*clock)();

// wait till time has gone past current time plus the amount we eanted to
// wait.  Note each click is approx. 55 milliseconds.

while(abs((*clock)() - now) < clicks){ Sleep(5); }

} // end Timer


/////////////////////////////////////////////////////////////////////////////

void Fade_Lights(void)
{
// this functions fades the lights by slowly decreasing the color values
// in all color registers

int index,pal_reg;
RGB_color color,color_1,color_2,color_3;

for (index=0; index<30; index++)
    {

    for (pal_reg=1; pal_reg<255; pal_reg++)
        {
        // get the color to fade

        Get_Palette_Register(pal_reg,(RGB_color_ptr)&color);

        if (color.red   > 5) color.red-=3;
        else
           color.red = 0;

        if (color.green > 5) color.green-=3;
        else
           color.green = 0;
        if (color.blue  > 5) color.blue-=3;
        else
           color.blue = 0;

        // set the color to a diminished intensity

        Set_Palette_Register(pal_reg,(RGB_color_ptr)&color);

        } // end for pal_reg

    // wait a bit

    Timer(2);
    } // end fade for

} // end Fade_Lights

//////////////////////////////////////////////////////////////////////////////

void Disolve(void )
{
       // disolve screen by ploting zillions of black pixels

       unsigned long index;

       for (index=0; index<=300000; index++)
       {
             Plot_Pixel_Fast(rand()%320, rand()%200, 0);

             if(!(index % 200)) { _redraw_screen(); }

       } // end Disolve
}
//////////////////////////////////////////////////////////////////////////////

void Melt(void )
{

// this function "melts" the screen by moving little worms at different speeds
// down the screen.  These worms change to the color thy are eating

int index,ticks=0;

// initialize the worms

for (index=0; index<160; index++)
    {

    worms[index].color   = Get_Pixel(index,0);
    worms[index].speed   = 3 + rand()%9;
    worms[index].y       = 0;
    worms[index].counter = 0;

    // draw the worm

    Plot_Pixel_Fast((index<<1),0,(char)worms[index].color);
    Plot_Pixel_Fast((index<<1),1,(char)worms[index].color);
    Plot_Pixel_Fast((index<<1),2,(char)worms[index].color);


    Plot_Pixel_Fast((index<<1)+1,0,(char)worms[index].color);
    Plot_Pixel_Fast((index<<1)+1,1,(char)worms[index].color);
    Plot_Pixel_Fast((index<<1)+1,2,(char)worms[index].color);

    } // end index
_redraw_screen();

// do screen melt

while(++ticks<1800)
     {

       if(!(index % 10)) _redraw_screen();

     // process each worm

     for (index=0; index<320; index++)
         {
         // is it time to move worm

         if (++worms[index].counter == worms[index].speed)
            {
            // reset counter
            worms[index].counter = 0;

            worms[index].color = Get_Pixel(index,worms[index].y+4);

            // has worm hit bottom?

            if (worms[index].y < 193)
               {

               Plot_Pixel_Fast((index<<1),worms[index].y,0);
               Plot_Pixel_Fast((index<<1),worms[index].y+1,(char)worms[index].color);
               Plot_Pixel_Fast((index<<1),worms[index].y+2,(char)worms[index].color);
               Plot_Pixel_Fast((index<<1),worms[index].y+3,(char)worms[index].color);

               Plot_Pixel_Fast((index<<1)+1,worms[index].y,0);
               Plot_Pixel_Fast((index<<1)+1,worms[index].y+1,(char)worms[index].color);
               Plot_Pixel_Fast((index<<1)+1,worms[index].y+2,(char)worms[index].color);
               Plot_Pixel_Fast((index<<1)+1,worms[index].y+3,(char)worms[index].color);

               worms[index].y++;

               } // end if worm isn't at bottom yet

            } // end if time to move worm

         } // end index

     // accelerate melt

     if (!(ticks % 500))
        {

        for (index=0; index<160; index++)
            worms[index].speed--;

        } // end if time to accelerate melt

     } // end while

} // end Melt

// M A I N ///////////////////////////////////////////////////////////////////

void main2(void )
{
int index,
    done=0,
    sel;

// set video mode to 320x200 256 color mode

Set_Mode(VGA256);
video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr
rom_char_set = (unsigned char far *)VGA_FONT_8X8;


PCX_Init((pcx_picture_ptr)&screen_fx);

PCX_Load("war.pcx", (pcx_picture_ptr)&screen_fx,1);

PCX_Show_Buffer((pcx_picture_ptr)&screen_fx);

PCX_Delete((pcx_picture_ptr)&screen_fx);

_settextposition(22,0);
printf("1 - Fade Lights.\n2 - Disolve.\n3 - Meltdown.");

// which special fx did user want to see

switch(getch())
      {
```

```
      case '1':  // dim lights
              {

              Fade_Lights();

              } break;

      case '2': // disolve screen
              {
              Disolve();

              } break;


      case '3': // melt screen
              {

              Melt();

              } break;

      } // end switch

// go back to text mode

Set_Mode(TEXT_MODE);

} // end main
```

SCALE.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
//#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>

#include "graph0.h" // include our graphics library

// D E F I N E S /////////////////////////////////////////////////////////////


// G L O B A L S /////////////////////////////////////////////////////////////

sprite object;
pcx_picture text_cells;

//////////////////////////////////////////////////////////////////////////////

// F U N C T I O N S /////////////////////////////////////////////////////////

void Scale_Sprite(sprite_ptr sprite,float scale)
```

```
{

// this function scale a sprite by computing the number of source pixels
// needed to satisfy the number of destination pixels

char far *work_sprite;
int work_offset=0,offset,x,y;
unsigned char data;
float y_scale_index,x_scale_step,y_scale_step,x_scale_index;

// set first source pixel

y_scale_index = 0;

// compute floating point step

y_scale_step = sprite_height/scale;
x_scale_step = sprite_width/scale;

// alias a pointer to sprite for ease of access

work_sprite = sprite->frames[sprite->curr_frame];

// compute offset of sprite in video buffer

offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;

// row by row scale object

for (y=0; y<(int)(scale); y++)
    {
    // copy the next row into the screen buffer using memcpy for speed

    x_scale_index=0;

    for (x=0; x<(int)scale; x++)
        {

        // test for transparent pixel i.e. 0, if not transparent then draw

        if ((data=work_sprite[work_offset+(int)x_scale_index]))
             double_buffer[offset+x] = data;

        x_scale_index+=(x_scale_step);

        } // end for x

    // using the floating scale_step, index to next source pixel

    y_scale_index+=y_scale_step;

    // move to next line in video buffer and in sprite bitmap buffer

    offset      += SCREEN_WIDTH;
    work_offset = sprite_width*(int)(y_scale_index);

    } // end for y

} // end Scale_Sprite

//////////////////////////////////////////////////////////////////////////////

void Clear_Double_Buffer()
{
// this function clears the double buffer, kinda crude, but G.E. (good enough)

_fmemset(double_buffer, 0, SCREEN_WIDTH * SCREEN_HEIGHT + 1);

} // end Clear_Double_Buffer

// M A I N ///////////////////////////////////////////////////////////////////
```

```
void main2(void)
{
int index,
    done=0;

float scale=64;

// set video mode to 320x200 256 color mode

Set_Mode(VGA256);
video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr
rom_char_set = (unsigned char far *)VGA_FONT_8X8;

// set sprite system size so that functions use correct sprite size

sprite_width = sprite_height = 64;

// initialize the pcx file that holds all the animation cells for net-tank

PCX_Init((pcx_picture_ptr)&text_cells);

// load the pcx file that holds the cells

PCX_Load("textures.pcx", (pcx_picture_ptr)&text_cells,1);

// PCX_Show_Buffer((pcx_picture_ptr)&text_cells);

Sprite_Init((sprite_ptr)&object,0,0,0,0,0,0);

// grap 4 interesting textures

PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,0,0,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,1,1,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,2,2,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,3,3,0);

// create some memory for the double buffer

Init_Double_Buffer();


// position object in center of screen

object.curr_frame = 0;
object.x          = 160-(sprite_width>>1);
object.y          = 100-(sprite_height>>1);

// clear the double buffer

Clear_Double_Buffer();

// show the user the scaled texture

Scale_Sprite((sprite_ptr)&object,scale);
Show_Double_Buffer((char *)double_buffer);

_settextposition(24,0);
printf("Q - Quit, < > - Scale, Space - Toggle.");

// main loop

while(!done)
     {
               FAST_CPU_WAIT(10);

     // has user hit a key?

     if (kbhit())
        {
        switch(getch())
```

```
              {
              case '.': // scale object larger
                   {
                   if (scale<180)
                      {
                      scale+=4;
                      object.x-=2;
                      object.y-=2;
                      } // end if ok to scale larger

                   } break;

              case ',': // scale object smaller
                   {
                   if (scale>4)
                      {
                      scale-=4;
                      object.x+=2;
                      object.y+=2;
                      } // end if ok to scale smaller

                   } break;

              case ' ': // go to next texture
                      {
                      // are we at the end?

                      if (++object.curr_frame==4)
                         object.curr_frame=0;

                      } break;

              case 'q': // let's go!
                      {
                      done=1;
                      } break;

              default:break;

              } // end switch


        // create a clean slate

        Clear_Double_Buffer();

        // scale the sprite and render into the double buffer

        Scale_Sprite((sprite_ptr)&object,scale);

        // show the double buffer

        Show_Double_Buffer((char *)double_buffer);

        _settextposition(24,0);
        printf("Q - Quit, < > - Scale, Space - Toggle.");

        }// end if

     } // end while

// delete the pcx file

PCX_Delete((pcx_picture_ptr)&text_cells);

// go back to text mode

Set_Mode(TEXT_MODE);

} // end main
```

AFIELD.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------

// I N C L U D E S ///////////////////////////////////////////////////////////

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
//#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>

#include "graph0.h" // include our graphics library

// D E F I N E S /////////////////////////////////////////////////////////////

#define NUM_STARS 30

// S T R U C T U R E S ///////////////////////////////////////////////////////

typedef struct star_typ
        {
        int x,y;    // position of star
        int vel;    // x - component of star velocity
        int color;  // color of star

        } star, *star_ptr;

// G L O B A L S /////////////////////////////////////////////////////////////

star stars[NUM_STARS]; // the star field
                                                          // 18.2 clicks/sec
sprite object;
pcx_picture ast_cells;

////////////////////////////////////////////////////////////////////////////////

// F U N C T I O N S /////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////

void Star_Field(void)
{

static int star_first=1;

// this function will create a panning 3-d star field with 3-planes, like
// looking out of the Enterprise

int index;

// test if we need to initialize star field i.e. first time function is being
// called

if (star_first)
```

```
   {
   // reset first time
   star_first=0;

   // initialize all the stars

   for (index=0; index<NUM_STARS; index++)
       {
       // initialize each star to a velocity, position and color

       stars[index].x      = rand()%320;
       stars[index].y      = rand()%180;

       // decide what star plane the star is in

       switch(rand()%3)
             {
             case 0: // plane 1- the farthest star plane
                  {
                  // set velocity and color

                  stars[index].vel = 2;
                  stars[index].color = 8;

                  } break;

             case 1: // plane 2-The medium distance star plane
                  {

                  stars[index].vel = 4;
                  stars[index].color = 7;

                  } break;

             case 2: // plane 3-The nearest star plane
                  {

                  stars[index].vel = 6;
                  stars[index].color = 15;

                  } break;

             } // end switch

       } // end for index

   } // end if first time
else
   { // must be nth time in, so do the usual

   // erase, move, draw

   for (index=0; index<NUM_STARS; index++)
       {

       if ( (stars[index].x+=stars[index].vel) >=320 )
          stars[index].x = 0;

       // draw

       Plot_Pixel_Fast_D(stars[index].x,stars[index].y,stars[index].color);

       } // end for index

   } // end else

} // end Star_Field

////////////////////////////////////////////////////////////////////////////////

void Scale_Sprite(sprite_ptr sprite,float scale)
```

```
{

// this function scale a sprite by computing the number of source pixels
// needed to satisfy the number of destination pixels

char far *work_sprite;
int work_offset=0,offset,x,y;
unsigned char data;
float y_scale_index,x_scale_step,y_scale_step,x_scale_index;

// set first source pixel

y_scale_index = 0;

// compute floating point step

y_scale_step = sprite_height/scale;
x_scale_step = sprite_width/scale;

// alias a pointer to sprite for ease of access

work_sprite = sprite->frames[sprite->curr_frame];

// compute offset of sprite in video buffer

offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;

// row by row scale object

for (y=0; y<(int)(scale); y++)
    {
    // copy the next row into the screen buffer using memcpy for speed

    x_scale_index=0;

    for (x=0; x<(int)scale; x++)
        {

        // test for transparent pixel i.e. 0, if not transparent then draw

        if ((data=work_sprite[work_offset+(int)x_scale_index]))
             double_buffer[offset+x] = data;

        x_scale_index+=(x_scale_step);

        } // end for x

    // using the floating scale_step, index to next source pixel

    y_scale_index+=y_scale_step;

    // move to next line in video buffer and in sprite bitmap buffer

    offset      += SCREEN_WIDTH;
    work_offset = sprite_width*(int)(y_scale_index);

    } // end for y

} // end Scale_Sprite

//////////////////////////////////////////////////////////////////////////////

void Clear_Double_Buffer(void)
{
// this function clears the double buffer, kinda crude, but G.E. (good enough)

_fmemset(double_buffer, 0, SCREEN_WIDTH * SCREEN_HEIGHT + 1);

} // end Clear_Double_Buffer

// M A I N /////////////////////////////////////////////////////////////////
```

```
void main2(void)
{
int index,
    done=0,dx=5,dy=4,ds=4;

float scale=5;

// set video mode to 320x200 256 color mode

Set_Mode(VGA256);
video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr
rom_char_set = (unsigned char far *)VGA_FONT_8X8;

// set sprite system size so that functions use correct sprite size

sprite_width = sprite_height = 47;

// initialize the pcx file that holds all the animation cells for net-tank

PCX_Init((pcx_picture_ptr)&ast_cells);

// load the pcx file that holds the cells

PCX_Load("asteroid.pcx", (pcx_picture_ptr)&ast_cells,1);

// create some memory for the double buffer

Init_Double_Buffer();

Sprite_Init((sprite_ptr)&object,0,0,0,0,0,0);

// load in frames of rotating asteroid

PCX_Grap_Bitmap((pcx_picture_ptr)&ast_cells,(sprite_ptr)&object,0,0,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&ast_cells,(sprite_ptr)&object,1,1,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&ast_cells,(sprite_ptr)&object,2,2,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&ast_cells,(sprite_ptr)&object,3,3,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&ast_cells,(sprite_ptr)&object,4,4,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&ast_cells,(sprite_ptr)&object,5,5,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&ast_cells,(sprite_ptr)&object,6,0,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&ast_cells,(sprite_ptr)&object,7,1,1);

// position object in center of screen

object.curr_frame  = 0;
object.x           = 160-(sprite_width>>1);
object.y           = 100-(sprite_height>>1);

// clear the double buffer

Clear_Double_Buffer();

// show the user the scaled texture

Scale_Sprite((sprite_ptr)&object,scale);
Show_Double_Buffer((char *)double_buffer);

// main loop

while(!kbhit())
     {
             FAST_CPU_WAIT(100);

     // scale asteroid

     scale+=ds;

     // test if asteroid is too big or too small

     if (scale>100 || scale < 5)
```

```
        {
        ds=-ds;
        scale+=ds;
        } //  end if we need to scale in other direction

     // move asteroid

     object.x+=dx;
     object.y+=dy;

     // test if object needs to bounch off wall

     if ((object.x + scale) > 310 || object.x < 10)
        {
        dx=-dx;
        object.x+=dx;
        } // end if hit a vertical boundary

     if ((object.y + scale) > 190 || object.y < 10)
        {
        dy=-dy;
        object.y+=dy;
        } // end if hit a horizontal boundary

     // rotate asteroid by 45

     if (++object.curr_frame==8)
         object.curr_frame=0;

     // create a clean slate

     Clear_Double_Buffer();

     // draw stars

     Star_Field();

     // scale the sprite and render into the double buffer

     Scale_Sprite((sprite_ptr)&object,scale);

     // show the double buffer

     Show_Double_Buffer((char *)double_buffer);

     } // end while

// delete the pcx file

PCX_Delete((pcx_picture_ptr)&ast_cells);

// go back to text mode

Set_Mode(TEXT_MODE);

} // end main
```

CHAP_08

VYREN.CPP

```
//--------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//--------------------------------------------
#include "stdafx.h"
```

```
#include "DOSEmu.h"
//------------------------------------------

// I N C L U D E S ///////////////////////////////////////////////////////////

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
//#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>

// #include <graph.h>

#include "graphics.h" // include our graphics library

// P R O T O T Y P E S ///////////////////////////////////////////////////////

void Create_Scale_Data_X(int scale, int far *row);

void Create_Scale_Data_Y(int scale, int *row);

void Build_Scale_Table(void);

void Scale_Sprite(sprite_ptr sprite,int scale);

void Clear_Double_Buffer(void);

// D E F I N E S /////////////////////////////////////////////////////////////

#define MAX_SCALE      200      // number of stars in star field
#define SPRITE_X_SIZE  80       // largest any bitmap can be
#define SPRITE_Y_SIZE  48       // the size of a sprite texture

// G L O B A L S /////////////////////////////////////////////////////////////

sprite object;                          // the generic sprite that will hold
                                        // the frames of the ship


pcx_picture text_cells;                 // the pcx file with the images

int *scale_table_y[MAX_SCALE+1];        // table with pre-computed scale indices

int far *scale_table_x[MAX_SCALE+1];    // table with pre-computed scale indices

//////////////////////////////////////////////////////////////////////////////

void Create_Scale_Data_X(int scale, int far *row)
{

// this function synthesizes the scaling of a texture sliver to all possible
// sizes and creates a huge look up table of the data.

int x;

float x_scale_index=0,
      x_scale_step;

// compute scale step or number of source pixels to map to destination/cycle

x_scale_step = (float)(sprite_width)/(float)scale;

x_scale_index+=x_scale_step;

for (x=0; x<scale; x++)
```

```
    {
    // place data into proper array position for later use

    row[x] = (int)(x_scale_index+.5);

    if  (row[x] > (SPRITE_X_SIZE-1)) row[x] = (SPRITE_X_SIZE-1);

    // next index please

    x_scale_index+=x_scale_step;

    } // end for x

} // end Create_Scale_Data_X

////////////////////////////////////////////////////////////////////////////////

void Create_Scale_Data_Y(int scale, int *row)
{

// this function synthesizes the scaling of a texture sliver to all possible
// sizes and creates a huge look up table of the data.

int y;

float y_scale_index=0,
      y_scale_step;

// compute scale step or number of source pixels to map to destination/cycle

y_scale_step = (float)(sprite_height)/(float)scale;

y_scale_index+=y_scale_step;

for (y=0; y<scale; y++)
    {
    // place data into proper array position for later use

    row[y] = ((int)(y_scale_index+.5)) * SPRITE_X_SIZE;

    if  (row[y] > (SPRITE_Y_SIZE-1)*SPRITE_X_SIZE) row[y] = (SPRITE_Y_SIZE-
1)*SPRITE_X_SIZE;

    // next index please

    y_scale_index+=y_scale_step;

    } // end for y

} // end Create_Scale_Data_Y

///////////////////////////////////////////////////////////////////////////////

void Build_Scale_Table(void)
{

// this function builds the scaler tables by computing the scale indices for all
// possible scales from 1-200 pixels high

int scale;


// allocate all the memory

for (scale=1; scale<=MAX_SCALE; scale++)
    {

    scale_table_y[scale] = (int *)malloc(scale*sizeof(int)+1);
    scale_table_x[scale] = (int far *)_fmalloc(scale*sizeof(int)+1);

    } // end for scale
```

```
// create the scale tables for both the X and Y axis

for (scale=1; scale<=MAX_SCALE; scale++)
    {

    // create the indices for this scale

    Create_Scale_Data_Y(scale, (int *)scale_table_y[scale]);
    Create_Scale_Data_X(scale, (int far *)scale_table_x[scale]);

    } // end for scale

} // end Build_Scale_Table

//////////////////////////////////////////////////////////////////////////////

void Scale_Sprite(sprite_ptr sprite,int scale)
{
// this function will scale the sprite (withoot clipping).  The scaling is done
// by looking into a pre-computed table that determines how how each vertical
// strip should be.  Then another table is used to compute how many of these
// vertical strips should be drawn based on the X scale of the object

char far *work_sprite;  // the sprite texture
int *row_y;             // pointer to the Y scale data (note: it is near)
int far *row_x;         // pointer to X scale data (note: it is far)

unsigned char pixel;    // the current textel

int x,                  // work variables
    y,
    column,
    work_offset,
    video_offset,
    video_start;

work_offset = 0;

// if object is too small, don't even bother rendering

if (scale<1) return;

// compute needed scaling data

row_y = scale_table_y[scale];
row_x = scale_table_x[scale];

// access the proper frame of the sprite

work_sprite = sprite->frames[sprite->curr_frame];

// compute where the starting video offset will always be

video_start = (sprite->y << 8) + (sprite->y << 6) + sprite->x;

// the images is drawn from left to right, top to bottom

for (x=0; x<scale; x++)
    {

    // re-cmpute next column address

    video_offset = video_start + x;

    // compute which column should be rendered based on X scale index

    column = row_x[x];

    // now do the column as we have always
    for (y=0; y<scale; y++)
        {

        // check for transparency

        pixel = work_sprite[work_offset+column];

        if (pixel)
            double_buffer[video_offset] = pixel;


        // index to next screen row and data offset in texture memory

        video_offset += SCREEN_WIDTH;
        work_offset  =  row_y[y];

        } // end for y

    } // end for x

} // end Scale_Sprite

//////////////////////////////////////////////////////////////////////////////

void Clear_Double_Buffer(void)
{

// take a guess?

_fmemset(double_buffer, 0, SCREEN_WIDTH * SCREEN_HEIGHT + 1);

} // end Clear_Double_Buffer

// M A I N ///////////////////////////////////////////////////////////////////

void main2(void)
{

printf("\n\n\n\nUse keys < > for fly, Q - for Quit");


// this main loads in the 12 frames of a pre-scanned image and rotates them
// while allowing the user to change the Z value of the object via the
// ',' and '.' keys

int done=0,                        // exit flag
    count=0,                       // used to count rtime till frame change
    scale=64;                      // current sprite scale

float scale_distance = 24000,      // arbitrary constants to make the flat texture
      view_distance = 256,         // scale properly in ray casted world

      x=0,                         // position of texture or ship in 3-SPACE
      y=0,
      z=1024;


// set video mode to 320x200 256 color mode

_setvideomode(_MRES256COLOR);
video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr
rom_char_set = (unsigned char far *)VGA_FONT_8X8;

sprite_width  = 80;
sprite_height = 48;

// create the look up tables for the scaler engine

Build_Scale_Table();

// initialize the pcx file that holds all the cells
```

```
PCX_Init((pcx_picture_ptr)&text_cells);

// load the pcx file that holds the cells

PCX_Load("vyrentxt.pcx", (pcx_picture_ptr)&text_cells,1);

// PCX_Show_Buffer((pcx_picture_ptr)&text_cells);

// create some memory for the double buffer

Init_Double_Buffer();

Sprite_Init((sprite_ptr)&object,0,0,0,0,0,0);

// load the 12 frames of the ship

PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,0,0,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,1,1,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,2,2,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,3,0,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,4,1,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,5,2,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,6,0,2);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,7,1,2);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,8,2,2);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,9,0,3);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,10,1,3);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,11,2,3);

// initialize the position of the ship

object.curr_frame = 0;
object.x          = 0;
object.y          = 0;

Clear_Double_Buffer();


// get user input and draw ship

while(!done)
     {
      FAST_CPU_WAIT(30);

     // has user hit keyboard

     if (kbhit())
        {
        switch(getch())
              {
              case '.': // move Z farther
                   {
                   z+=16;
                   } break;

              case ',': // move Z closer
                   {
                   z-=16;

                   // don't let object get too close

                   if (z<256)
                      z=256;

                   } break;

              case 'q': // exit program
                      {
                      done=1;
                      } break;
```

```
                default:break;

                } // end switch

            } // end if


            // compute the size of the bitmap

            scale = (int)( scale_distance/z );

            // based on the size of the bitmap, compute the perspective X and Y

            object.x = (int)((float)x*view_distance / (float)z) + 160 - (scale>>1);
            object.y = 100 - (((int)((float)y*view_distance / (float)z) + (scale>>1)) );


            // increment frame counter to next frame

            if (++count==2)
               {
               count=0;

               if (++object.curr_frame==12)
                   object.curr_frame=0;

               } // end if time to change frames


            // blank out the double buffer

            Clear_Double_Buffer();

            // scale the sprite to it's proper size

            Scale_Sprite((sprite_ptr)&object,scale);

            Show_Double_Buffer((char *)double_buffer);


            // show user some info

            _settextposition(24,0);
            printf("Z Coordinate is %f",z);

         } // end while

// delete the pcx file

PCX_Delete((pcx_picture_ptr)&text_cells);

// go back to text mode

_setvideomode(_DEFAULTMODE);

} // end main
```

FINVYREN.CPP

```
//--------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//--------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//--------------------------------------------
```

```
// I N C L U D E S ///////////////////////////////////////////////////////////

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
// #include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>

// #include <graph.h>

#include "graphics.h" // include our graphics library

// P R O T O T Y P E S ///////////////////////////////////////////////////////

void Create_Scale_Data_X(int scale, int far *row);

void Create_Scale_Data_Y(int scale, int *row);

void Build_Scale_Table(void);

void Scale_Sprite(sprite_ptr sprite,int scale);

void Clear_Double_Buffer(void);

void Timer(int clicks);

void Init_Stars(void);

void Move_Stars(void);

void Draw_Stars(void);

// D E F I N E S /////////////////////////////////////////////////////////////

#define NUM_STARS      50       // number of stars in star field
#define MAX_SCALE      200      // largest any bitmap can be
#define SPRITE_X_SIZE  80       // the size of a sprite texture
#define SPRITE_Y_SIZE  48

// S T R U C T U R E S ///////////////////////////////////////////////////////

// this is a star

typedef struct star_typ
        {

        int x,y;        // postion of star
        int xv,yv;      // velocity of star
        int xa,ya;      // acceleration of star
        int color;      // color of star
        int clock;      // number of ticks star has been alive
        int acc_time;   // this is the number of ticks to count before accelerating
        int acc_count;  // the accleration counter

        } star, *star_ptr;

// G L O B A L S /////////////////////////////////////////////////////////////

// unsigned int far *clock = (unsigned int far *)0x0000046C; // pointer to internal
                                                          // 18.2 clicks/sec
// USAGE: ULONG now = (*clock)();
ULONG (*clock)() = PMEM_0x0000046C;

sprite object;                          // the generic sprite that will hold
                                        // the frames of the ship
```

```
pcx_picture text_cells;                  // the pcx file with the images

int *scale_table_y[MAX_SCALE+1];         // table with pre-computed scale indices

int far *scale_table_x[MAX_SCALE+1];     // table with pre-computed scale indices

star star_field[NUM_STARS];              // the star field

//////////////////////////////////////////////////////////////////////////////

void Timer(int clicks)
{
//------------------------------------------
// EACH CLICK IS APPROX. 55 MILLISECONDS.
// YOU MAY USE Sleep() ... BUT IT'S NOT A PLAIN DOS FEATURE...
/*
Sleep(clicks*55);
return;
*/
//------------------------------------------


// this function uses the internal time keeper timer i.e. the one that goes
// at 18.2 clicks/sec to to a time delay.  You can find a 32 bit value of
// this timer at 0000:046Ch

unsigned int now;

// get current time

now = (*clock)();

// wait till time has gone past current time plus the amount we eanted to
// wait.  Note each click is approx. 55 milliseconds.

while(abs((*clock)() - now) < clicks){ Sleep(5); }

} // end Timer


//////////////////////////////////////////////////////////////////////////////

void Init_Stars(void)
{

// this function initializes the star field data structure when the system
// is started

int index,divisor;

for (index=0; index<NUM_STARS; index++)
    {

    star_field[index].x      = 150 + rand() % 20;
    star_field[index].y      = 90  + rand() % 20;


    if (rand()%2==1)
       star_field[index].xv     = -4 + -2 * rand() % 3;
    else
       star_field[index].xv     = 4 + 2 * rand() % 3;

    if (rand()%2==1)
       star_field[index].yv     = -4 + -2 * rand() % 3;
    else
       star_field[index].yv     = 4 + 2 * rand() % 3;


    divisor = 1 + rand()%3;
    star_field[index].xa     = star_field[index].xv/divisor;
    star_field[index].ya     = star_field[index].yv/divisor;

    star_field[index].color     = 7;
    star_field[index].clock     = 0;

    star_field[index].acc_time  = 1 + rand() % 3;
    star_field[index].acc_count = 0;


    } // end index

} // end Init_Stars

//////////////////////////////////////////////////////////////////////////////

void Move_Stars(void)
{
// this function moves the stars, and tests if a star has gone off the screen
// if so, the function starts the star over again

int index,divisor;

for (index=0; index<NUM_STARS; index++)
    {

    star_field[index].x += star_field[index].xv;
    star_field[index].y += star_field[index].yv;

    // test if star is off screen

    if (star_field[index].x >= SCREEN_WIDTH  || star_field[index].x < 0 ||
        star_field[index].y >= SCREEN_HEIGHT || star_field[index].y < 0)
       {

       // restart the star

       star_field[index].x      = 150 + rand() % 20;
       star_field[index].y      = 90  + rand() % 20;

       if (rand()%2==1)
          star_field[index].xv     = -4 + -2 * rand() % 3;
       else
          star_field[index].xv     = 4 + 2 * rand() % 3;

       if (rand()%2==1)
          star_field[index].yv     = -4 + -2 * rand() % 3;
       else
          star_field[index].yv     = 4 + 2 * rand() % 3;


       divisor = 1 + rand()%3;

       star_field[index].xa     = star_field[index].xv/divisor;
       star_field[index].ya     = star_field[index].yv/divisor;

       star_field[index].color     = 7;
       star_field[index].clock     = 0;

       star_field[index].acc_time  = 1 + rand() % 3;
       star_field[index].acc_count = 0;

       } // end if

    // test if it's time to accelerate

    if (++star_field[index].acc_count==star_field[index].acc_time)
       {
       // reset counter
       star_field[index].acc_count=0;

       // accelerate

       star_field[index].xv += star_field[index].xa;
       star_field[index].yv += star_field[index].ya;

       } // end if time to accelerate

    // test if it's time to change color

    if (++star_field[index].clock > 5)
       {
       star_field[index].color = 8;

       } // end if > 10
    else
    if (star_field[index].clock > 10)
       {
       star_field[index].color =255;

       } // end if > 20
    else
    if (star_field[index].clock > 25)
       {
       star_field[index].color = 255;

       } // end if > 25

    } // end for index

} // end Move_Stars

//////////////////////////////////////////////////////////////////////////////

void Draw_Stars(void)
{
// this function draws the stars into the double buffer

int index;

for (index=0; index<NUM_STARS; index++)
    {

    Plot_Pixel_Fast_D(star_field[index].x,star_field[index].y,(unsigned
char)star_field[index].color);

    } // end for index

} // end Draw_Stars

//////////////////////////////////////////////////////////////////////////////

void Create_Scale_Data_X(int scale, int far *row)
{

// this function synthesizes the scaling of a texture sliver to all possible
// sizes and creates a huge look up table of the data.

int x;

float x_scale_index=0,
      x_scale_step;

// compute scale step or number of source pixels to map to destination/cycle

x_scale_step = (float)(sprite_width)/(float)scale;

x_scale_index+=x_scale_step;

for (x=0; x<scale; x++)
```

```
    {
    // place data into proper array position for later use

    row[x] = (int)(x_scale_index+.5);

    if  (row[x] > (SPRITE_X_SIZE-1)) row[x] = (SPRITE_X_SIZE-1);

    // next index please

    x_scale_index+=x_scale_step;

    } // end for x

} // end Create_Scale_Data_X

//////////////////////////////////////////////////////////////////////////////

void Create_Scale_Data_Y(int scale, int *row)
{

// this function synthesizes the scaling of a texture sliver to all possible
// sizes and creates a huge look up table of the data.

int y;

float y_scale_index=0,
      y_scale_step;

// compute scale step or number of source pixels to map to destination/cycle

y_scale_step = (float)(sprite_height)/(float)scale;

y_scale_index+=y_scale_step;

for (y=0; y<scale; y++)
    {
    // place data into proper array position for later use

    row[y] = ((int)(y_scale_index+.5)) * SPRITE_X_SIZE;

    if  (row[y] > (SPRITE_Y_SIZE-1)*SPRITE_X_SIZE) row[y] = (SPRITE_Y_SIZE-
1)*SPRITE_X_SIZE;

    // next index please

    y_scale_index+=y_scale_step;

    } // end for y

} // end Create_Scale_Data_Y

/////////////////////////////////////////////////////////////////////////////

void Build_Scale_Table(void)
{

// this function builds the scaler tables by computing the scale indices for all
// possible scales from 1-200 pixels high

int scale;


// allocate all the memory

for (scale=1; scale<=MAX_SCALE; scale++)
    {

    scale_table_y[scale] = (int *)malloc(scale*sizeof(int)+1);
    scale_table_x[scale] = (int far *)_fmalloc(scale*sizeof(int)+1);

    } // end for scale
```

```
// create the scale tables for both the X and Y axis

for (scale=1; scale<=MAX_SCALE; scale++)
    {

    // create the indices for this scale

    Create_Scale_Data_Y(scale, (int *)scale_table_y[scale]);
    Create_Scale_Data_X(scale, (int far *)scale_table_x[scale]);

    } // end for scale

} // end Build_Scale_Table

//////////////////////////////////////////////////////////////////////////////

void Scale_Sprite(sprite_ptr sprite,int scale)
{
// this function will scale the sprite (withoot clipping).  The scaling is done
// by looking into a pre-computed table that determines how how each vertical
// strip should be.  Then another table is used to compute how many of these
// vertical strips should be drawn based on the X scale of the object

char far *work_sprite;  // the sprite texture
int *row_y;             // pointer to the Y scale data (note: it is near)
int far *row_x;         // pointer to X scale data (note: it is far)

unsigned char pixel;    // the current textel

int x = 0,                  // work variables
    y = 0,
    column = 0,
    work_offset = 0,
    video_offset = 0,
    video_start = 0;


// if object is too small, don't even bother rendering

if (scale<1) return;

// compute needed scaling data

row_y = scale_table_y[scale];
row_x = scale_table_x[scale];

// access the proper frame of the sprite

work_sprite = sprite->frames[sprite->curr_frame];

// compute where the starting video offset will always be

video_start = (sprite->y << 8) + (sprite->y << 6) + sprite->x;

// the images is drawn from left to right, top to bottom

for (x=0; x<scale; x++)
    {

    // re-compute next column address

    video_offset = video_start + x;

    // compute which column should be rendered based on X scale index

    column = row_x[x];

    // now do the column as we have always

    for (y=0; y<scale; y++)
```

```
        {

        // check for transparency

        pixel = work_sprite[work_offset+column];

        if (pixel)
            double_buffer[video_offset] = pixel;


        // index to next screen row and data offset in texture memory

        video_offset += SCREEN_WIDTH;
        work_offset  =  row_y[y];

        } // end for y

    } // end for x

} // end Scale_Sprite

//////////////////////////////////////////////////////////////////////////////

void Clear_Double_Buffer(void)
{

// take a guess?

_fmemset(double_buffer, 0, SCREEN_WIDTH * SCREEN_HEIGHT + 1);

} // end Clear_Double_Buffer

// M A I N ///////////////////////////////////////////////////////////////////

void main2(void)
{
printf("\n\n\n\nUse ARROWS keys at NUMPAD and q for exit.\n");

// this main places the ship into a star field and allows the user to fly around

int done=0,                     // exit flag
    scale=64,
    direction=6;                // the direction of the ship (current frame)

float scale_distance = 24000, // arbitrary constants to make the flat texture
      view_distance = 256,    // scale properly in ray casted world

      x=0,                      // position of texture or ship in 3-SPACE
      y=0,
      z=1024,
      xv=0,zv=0,                // velocity of ship in X-Z plane
      angle=180,                // angle of ship
      ship_speed=10;            // magnitude of ships speed

// set video mode to 320x200 256 color mode

_setvideomode(_MRES256COLOR);
video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr

// all sprites will have this size

sprite_width  = 80;
sprite_height = 48;

// create the look up tables for the scaler engine

Build_Scale_Table();

// initialize the pcx file that holds all the cells

PCX_Init((pcx_picture_ptr)&text_cells);
```

```
// load the pcx file that holds the cells

PCX_Load("vyrentxt.pcx", (pcx_picture_ptr)&text_cells,1);

// PCX_Show_Buffer((pcx_picture_ptr)&text_cells);

// create some memory for the double buffer

Init_Double_Buffer();

// initialize the star field

Init_Stars();

// set the ships direction and velocity up

angle=direction*30+90;

xv = (float)(ship_speed*cos(3.14159*angle/180));
zv = (float)(ship_speed*sin(3.14159*angle/180));

Sprite_Init((sprite_ptr)&object,0,0,0,0,0,0);

// load the 12 frames of the ship

PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,0,0,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,1,1,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,2,2,0);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,3,0,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,4,1,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,5,2,1);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,6,0,2);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,7,1,2);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,8,2,2);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,9,0,3);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,10,1,3);
PCX_Grap_Bitmap((pcx_picture_ptr)&text_cells,(sprite_ptr)&object,11,2,3);

// initialize the position of the ship

object.curr_frame = 0;
object.x          = 0;
object.y          = 0;

Clear_Double_Buffer();


// get user input and draw ship

while(!done)
     {

     // has user hit keyboard

     if (kbhit())
        {
        switch(getch())
              {
              case '4': // turn ship left
                   {

                   if (++direction==12)
                      {

                      direction=0;

                      } // end if wrap around
                   } break;

              case '6': // turn ship right
```

```
           {

           if (--direction < 0)
              {

              direction=11;

              } // end if wrap around

           } break;

      case '8': // speed ship up
              {
              if (++ship_speed > 20)
                 ship_speed=20;

              } break;

      case '2': // slow ship down
              {
              if (--ship_speed < 0)
                 ship_speed=0;

              } break;

      case 'q': // exit program
              {
              done=1;
              } break;

      default:break;

      } // end switch

// reset velocity and direction vectors

angle=direction*30+90;

xv = (float)(ship_speed*cos(3.14159*angle/180));
zv = (float)(ship_speed*sin(3.14159*angle/180));

} // end if

// translate the ship each cycle

x+=xv;
z+=zv;

// bound to hither plane

if (z<256)
   z=256;

// compute the size of the bitmap

scale = (int)( scale_distance/z );

// based on the size of the bitmap, compute the perspective X and Y

object.x = (int)((float)x*view_distance / (float)z) + 160 - (scale>>1);
object.y = 100 - (((int)((float)y*view_distance / (float)z) + (scale>>1)) );

// bound to screen edges

if (object.x < 0 )
    object.x = 0;
else
if (object.x+scale >= SCREEN_WIDTH)
    object.x = SCREEN_WIDTH-scale;
```

```
        if (object.y < 0 )
            object.y = 0;
        else
        if (object.y+scale >= SCREEN_HEIGHT)
            object.y = SCREEN_HEIGHT-scale;


        // set current frame

        object.curr_frame = direction;

        // blank out the double buffer

        Clear_Double_Buffer();

        Move_Stars();

        Draw_Stars();

        // scale the sprite to it's proper size

        Scale_Sprite((sprite_ptr)&object,scale);

        Show_Double_Buffer((char *)double_buffer);

        // show user some info

        _settextposition(23,0);
        printf("Position=(%4.2f,%4.2f,%4.2f)   ",x,y,z);

       // slow things down a bit

       Timer(1);

     } // end while

// delete the pcx file

PCX_Delete((pcx_picture_ptr)&text_cells);

// go back to text mode

_setvideomode(_DEFAULTMODE);

} // end main
```

GRAPHICS.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------

// I N C L U D E S ///////////////////////////////////////////////////////////

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
//#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
```

```
#include <string.h>
#include "graphics.h"

// G L O B A L S  ////////////////////////////////////////////////////////////

unsigned char far *video_buffer = (unsigned char *)(char far *)0xA0000000L; // vram byte
ptr
unsigned int far *video_buffer_w= (unsigned int *)(int far *)0xA0000000L;  // vram word ptr
unsigned char far *rom_char_set = (unsigned char *)(char far *)0xF000FA6EL; // rom
characters 8x8
unsigned char far *double_buffer = NULL;
unsigned int sprite_width = SPRITE_WIDTH; // default height and width of sprite
unsigned int sprite_height = SPRITE_HEIGHT;


// F U N C T I O N S /////////////////////////////////////////////////////////


void Blit_Char(int xc,int yc,char c,int color,int trans_flag)
{
// this function uses the rom 8x8 character set to blit a character on the
// video screen, notice the trick used to extract bits out of each character
// byte that comprises a line

int offset,x,y;
unsigned char data;
char far *work_char;
unsigned char bit_mask = 0x80;

// compute starting offset in rom character lookup table

work_char = (char *)rom_char_set + c * CHAR_HEIGHT;

// compute offset of character in video buffer

offset = (yc << 8) + (yc << 6) + xc;

for (y=0; y<CHAR_HEIGHT; y++)
    {
    // reset bit mask

    bit_mask = 0x80;

    for (x=0; x<CHAR_WIDTH; x++)
        {
        // test for transparent pixel i.e. 0, if not transparent then draw

        if ((*work_char & bit_mask))
             video_buffer[offset+x] = color;

        else if (!trans_flag)  // takes care of transparency
            video_buffer[offset+x] = 0;

        // shift bit mask

        bit_mask = (bit_mask>>1);

        } // end for x

    // move to next line in video buffer and in rom character data area

    offset      += SCREEN_WIDTH;
    work_char++;

    } // end for y

} // end Blit_Char

//////////////////////////////////////////////////////////////////////////////

void Blit_String(int x,int y,int color, char *string,int trans_flag)
```

```
{
// this function blits an entire string on the screen with fixed spacing
// between each character.  it calls blit_char.

int index;

for (index=0; string[index]!=0; index++)
     {

     Blit_Char(x+(index<<3),y,string[index],color,trans_flag);

     } /* end while */

      _redraw_screen();

} /* end Blit_String */

//////////////////////////////////////////////////////////////////////////////

void Delay(int t)
{

float x = 1;

while(t-->0)
     x=cos(x);

} // end Delay

//////////////////////////////////////////////////////////////////////////////

void Set_Palette_Register(int index, RGB_color_ptr color)
{

// this function sets a single color look up table value indexed by index
// with the value in the color structure

// tell VGA card we are going to update a pallete register

_outp(PALETTE_MASK,0xff);

// tell vga card which register we will be updating

_outp(PALETTE_REGISTER_WR, index);

// now update the RGB triple, note the same port is used each time

_outp(PALETTE_DATA,color->red);
_outp(PALETTE_DATA,color->green);
_outp(PALETTE_DATA,color->blue);

} // end Set_Palette_Color

///////////////////////////////////////////////////////////////////////////////

void Get_Palette_Register(int index, RGB_color_ptr color)
{

// this function gets the data out of a color lookup regsiter and places it
// into color

// set the palette mask register

_outp(PALETTE_MASK,0xff);

// tell vga card which register we will be reading

_outp(PALETTE_REGISTER_RD, index);

// now extract the data
color->red   = _inp(PALETTE_DATA);
color->green = _inp(PALETTE_DATA);
color->blue  = _inp(PALETTE_DATA);

} // end Get_Palette_Color

//////////////////////////////////////////////////////////////////////////////

void PCX_Init(pcx_picture_ptr image)
{
// this function allocates the buffer region needed to load a pcx file

if (!(image->buffer = (char far *)_fmalloc(SCREEN_WIDTH * SCREEN_HEIGHT + 1)))

   printf("\ncouldn't allocate screen buffer");

} // end PCX_Init

//////////////////////////////////////////////////////////////////////////////

void Plot_Pixel_Fast(int x,int y,unsigned char color)
{

// plots the pixel in the desired color a little quicker using binary shifting
// to accomplish the multiplications

// use the fact that 320*y = 256*y + 64*y = y<<8 + y<<6

video_buffer[((y<<8) + (y<<6)) + x] = color;

} // end Plot_Pixel_Fast

//////////////////////////////////////////////////////////////////////////////

void Plot_Pixel_Fast_D(int x,int y,unsigned char color)
{

// plots the pixel in the desired color a little quicker using binary shifting
// to accomplish the multiplications

// use the fact that 320*y = 256*y + 64*y = y<<8 + y<<6

double_buffer[((y<<8) + (y<<6)) + x] = color;

} // end Plot_Pixel_Fast_D

//////////////////////////////////////////////////////////////////////////////

unsigned char Get_Pixel(int x,int y)
{

// gets the pixel from the screen

return video_buffer[((y<<8) + (y<<6)) + x];

} // end Get_Pixel

//////////////////////////////////////////////////////////////////////////////

unsigned char Get_Pixel_D(int x,int y)
{

// gets the pixel from the screen

return double_buffer[((y<<8) + (y<<6)) + x];

} // end Get_Pixel_D

//////////////////////////////////////////////////////////////////////////////

void PCX_Delete(pcx_picture_ptr image)
```

```
{
// this function de-allocates the buffer region used for the pcx file load

_ffree(image->buffer);

} // end PCX_Delete

//////////////////////////////////////////////////////////////////////////////

void PCX_Load(char *filename, pcx_picture_ptr image,int enable_palette)
{
// this function loads a pcx file into a picture structure, the actual image
// data for the pcx file is decompressed and expanded into a secondary buffer
// within the picture structure, the separate images can be grabbed from this
// buffer later.  also the header and palette are loaded

FILE *fp /*, *fopen()*/;
int num_bytes,index;
long count;
unsigned char data;
char far *temp_buffer;

// open the file

fp = fopen(filename,"rb");

// load the header

temp_buffer = (char far *)image;

for (index=0; index<128; index++)
    {
    temp_buffer[index] = getc(fp);
    } // end for index

// load the data and decompress into buffer

count=0;

while(count<=SCREEN_WIDTH * SCREEN_HEIGHT)
     {
     // get the first piece of data

     data = getc(fp);

     // is this a rle?

     if (data>=192 && data<=255)
        {
        // how many bytes in run?

        num_bytes = data-192;

        // get the actual data for the run

        data  = getc(fp);

        // replicate data in buffer num_bytes times

        while(num_bytes-->0)
             {
             image->buffer[count++] = data;

             } // end while

        } // end if rle
     else
        {
        // actual data, just copy it into buffer at next location

        image->buffer[count++] = data;
        } // end else not rle

     } // end while

// move to end of file then back up 768 bytes i.e. to begining of palette

fseek(fp,-768L,SEEK_END);

// load the pallete into the palette

for (index=0; index<256; index++)
    {
    // get the red component

    image->palette[index].red   = (getc(fp) >> 2);

    // get the green component

    image->palette[index].green = (getc(fp) >> 2);

    // get the blue component

    image->palette[index].blue  = (getc(fp) >> 2);

    } // end for index

fclose(fp);

// change the palette to newly loaded palette if commanded to do so

if (enable_palette)
   {

   for (index=0; index<256; index++)
       {

       Set_Palette_Register(index,(RGB_color_ptr)&image->palette[index]);

       } // end for index

   } // end if change palette

} // end PCX_Load

//////////////////////////////////////////////////////////////////////////////

void PCX_Show_Buffer(pcx_picture_ptr image)
{
// just copy he pcx buffer into the video buffer

// _fmemcpy((char far *)video_buffer,
//          (char far *)image->buffer,SCREEN_WIDTH*SCREEN_HEIGHT);

 _fmemcpy((char far *)video_buffer,
      (char far *)image->buffer,SCREEN_WIDTH*SCREEN_HEIGHT);

      _redraw_screen();

//-----------------------------------------
// NO ANY ASM
//-----------------------------------------
/*
char far *data;

data = image->buffer;

_asm
   {
   push ds
   les di, video_buffer
   lds si, data
   mov cx,320*200/2
   cld
   rep movsw
   pop ds
   }
*/
} // end PCX_Show_Picture

//////////////////////////////////////////////////////////////////////////////

void Show_Double_Buffer(char far *buffer)
{
// copy the double buffer into the video buffer

_fmemcpy((char far *)video_buffer,
       (char far *)buffer,SCREEN_WIDTH*SCREEN_HEIGHT);

      _redraw_screen();

//-----------------------------------------
// NO ANY ASM
//-----------------------------------------
/*
_asm
   {
   push ds
   les di, video_buffer
   lds si, buffer
   mov cx,320*200/2
   cld
   rep movsw
   pop ds
   }
*/
} // end Show_Double_Buffer

//////////////////////////////////////////////////////////////////////////////

void Init_Double_Buffer(void)
{

double_buffer = (unsigned char *)(char far *)_fmalloc(SCREEN_WIDTH * SCREEN_HEIGHT + 1);

_fmemset(double_buffer, 0, SCREEN_WIDTH * SCREEN_HEIGHT + 1);

} // end Init_Double_Buffer

//////////////////////////////////////////////////////////////////////////////

void Sprite_Init(sprite_ptr sprite,int x,int y,int ac,int as,int mc,int ms)
{
// this function initializes a sprite with the sent data

int index;

sprite->x            = x;
sprite->y            = y;
sprite->x_old        = x;
sprite->y_old        = y;
sprite->width        = sprite_width;
sprite->height       = sprite_height;
sprite->anim_clock   = ac;
sprite->anim_speed   = as;
sprite->motion_clock = mc;
sprite->motion_speed = ms;
sprite->curr_frame   = 0;
sprite->state        = SPRITE_DEAD;
sprite->num_frames   = 0;
sprite->background   = (char far *)_fmalloc(sprite_width * sprite_height+1);
```

```
// set all bitmap pointers to null

for (index=0; index<MAX_SPRITE_FRAMES; index++)
    sprite->frames[index] = NULL;

} // end Sprite_Init

//////////////////////////////////////////////////////////////////////////////

void Sprite_Delete(sprite_ptr sprite)
{
// this function deletes all the memory associated with a sprire

int index;

_ffree(sprite->background);

// now de-allocate all the animation frames

for (index=0; index<MAX_SPRITE_FRAMES; index++)
    _ffree(sprite->frames[index]);

} // end Sprite_Delete


//////////////////////////////////////////////////////////////////////////////

void PCX_Grap_Bitmap(pcx_picture_ptr image,
                    sprite_ptr sprite,
                    int sprite_frame,
                    int grab_x, int grab_y)

{
// this function will grap a bitmap from the pcx frame buffer. it uses the
// convention that the 320x200 pixel matrix is sub divided into a smaller
// matrix of nxn adjacent squares

int x_off,y_off, x,y, index;
char far *sprite_data;

// first allocate the memory for the sprite in the sprite structure

sprite->frames[sprite_frame] = (char far *)_fmalloc(sprite_width * sprite_height + 1);

// create an alias to the sprite frame for ease of access

sprite_data = sprite->frames[sprite_frame];

// now load the sprite data into the sprite frame array from the pcx picture

x_off = (sprite_width+1)  * grab_x + 1;
y_off = (sprite_height+1) * grab_y + 1;

// compute starting y address

y_off = y_off * 320;

for (y=0; y<sprite_height; y++)
    {

    for (x=0; x<sprite_width; x++)
        {

        // get the next byte of current row and place into next position in
        // sprite frame data buffer

        sprite_data[y*sprite_width + x] = image->buffer[y_off + x_off + x];

        } // end for x

        // move to next line of picture buffer
        y_off+=320;

    } // end for y

// increment number of frames

sprite->num_frames++;

// done!, let's bail!

} // end PCX_Grap_Bitmap

//////////////////////////////////////////////////////////////////////////////

void Behind_Sprite(sprite_ptr sprite)
{

// this function scans the background behind a sprite so that when the sprite
// is draw, the background isnn'y obliterated

char far *work_back;
int work_offset=0,offset,y;

// alias a pointer to sprite background for ease of access

work_back = sprite->background;

// compute offset of background in video buffer

offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;

for (y=0; y<sprite_height; y++)
    {
    // copy the next row out off screen buffer into sprite background buffer

    _fmemcpy((char far *)&work_back[work_offset],
             (char far *)&double_buffer[offset],
             sprite_width);

    // move to next line in video buffer and in sprite background buffer

    offset      += SCREEN_WIDTH;
    work_offset += sprite_width;

    } // end for y

} // end Behind_Sprite

//////////////////////////////////////////////////////////////////////////////

void Erase_Sprite(sprite_ptr sprite)
{
// replace the background that was behind the sprite

// this function replaces the background that was saved from where a sprite
// was going to be placed

char far *work_back;
int work_offset=0,offset,y;

// alias a pointer to sprite background for ease of access

work_back = sprite->background;

// compute offset of background in video buffer

offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;

for (y=0; y<sprite_height; y++)
    {
```

```
    // copy the next row out off screen buffer into sprite background buffer

    _fmemcpy((char far *)&double_buffer[offset],
             (char far *)&work_back[work_offset],
             sprite_width);

    // move to next line in video buffer and in sprite background buffer

    offset      += SCREEN_WIDTH;
    work_offset += sprite_width;

    } // end for y


} // end Erase_Sprite

//////////////////////////////////////////////////////////////////////////////

void Draw_Sprite(sprite_ptr sprite)
{

// this function draws a sprite on the screen row by row very quickly
// note the use of shifting to implement multplication

char far *work_sprite;
int work_offset=0,offset,x,y;
unsigned char data;

// alias a pointer to sprite for ease of access

work_sprite = sprite->frames[sprite->curr_frame];

// compute offset of sprite in video buffer

offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;

for (y=0; y<sprite_height; y++)
    {
    // copy the next row into the screen buffer using memcpy for speed

    for (x=0; x<sprite_width; x++)
        {

        // test for transparent pixel i.e. 0, if not transparent then draw

        if ((data=work_sprite[work_offset+x]))
             double_buffer[offset+x] = data;

        } // end for x

    // move to next line in video buffer and in sprite bitmap buffer

    offset      += SCREEN_WIDTH;
    work_offset += sprite_width;

    } // end for y

} // end Draw_Sprite

//////////////////////////////////////////////////////////////////////////////

void Behind_Sprite_VB(sprite_ptr sprite)
{

// this function scans the background behind a sprite so that when the sprite
// is draw, the background isnn'y obliterated

char far *work_back;
int work_offset=0,offset,y;

// alias a pointer to sprite background for ease of access
```

```
work_back = sprite->background;

// compute offset of background in video buffer

offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;

for (y=0; y<sprite_height; y++)
    {
    // copy the next row out off screen buffer into sprite background buffer

    _fmemcpy((char far *)&work_back[work_offset],
             (char far *)&video_buffer[offset],
             sprite_width);

    // move to next line in video buffer and in sprite background buffer

    offset      += SCREEN_WIDTH;
    work_offset += sprite_width;

    } // end for y

      _redraw_screen();

} // end Behind_Sprite_VB

//////////////////////////////////////////////////////////////////////////////

void Erase_Sprite_VB(sprite_ptr sprite)
{
// replace the background that was behind the sprite

// this function replaces the background that was saved from where a sprite
// was going to be placed

char far *work_back;
int work_offset=0,offset,y;

// alias a pointer to sprite background for ease of access

work_back = sprite->background;

// compute offset of background in video buffer

offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;

for (y=0; y<sprite_height; y++)
    {
    // copy the next row out off screen buffer into sprite background buffer

    _fmemcpy((char far *)&video_buffer[offset],
             (char far *)&work_back[work_offset],
             sprite_width);

    // move to next line in video buffer and in sprite background buffer

    offset      += SCREEN_WIDTH;
    work_offset += sprite_width;

    } // end for y

      _redraw_screen();

} // end Erase_Sprite_VB

//////////////////////////////////////////////////////////////////////////////

void Draw_Sprite_VB(sprite_ptr sprite)
{

// this function draws a sprite on the screen row by row very quickly
// note the use of shifting to implement multplication

char far *work_sprite;
int work_offset=0,offset,x,y;
unsigned char data;

// alias a pointer to sprite for ease of access

work_sprite = sprite->frames[sprite->curr_frame];

// compute offset of sprite in video buffer

offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;

for (y=0; y<sprite_height; y++)
    {
    // copy the next row into the screen buffer using memcpy for speed

    for (x=0; x<sprite_width; x++)
        {

        // test for transparent pixel i.e. 0, if not transparent then draw

        if ((data=work_sprite[work_offset+x]))
             video_buffer[offset+x] = data;

        } // end for x

    // move to next line in video buffer and in sprite bitmap buffer

    offset      += SCREEN_WIDTH;
    work_offset += sprite_width;

    } // end for y

      _redraw_screen();

} // end Draw_Sprite_VB
```

CHAP_09

SOUND.CPP

```
//-----------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//-----------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//-----------------------------------------

// I N C L U D E S ///////////////////////////////////////////////////////////

#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
//#include <bios.h>
#include <fcntl.h>

// G L O B A L S  ////////////////////////////////////////////////////////////

char __far *driver_ptr;
unsigned version;
char __huge *data_ptr;
unsigned ct_voice_status;
unsigned char *current_sound_play = NULL;
unsigned time_sound_end_play = 0;

// F U N C T I O N S /////////////////////////////////////////////////////////

void Voc_Get_Version(void)
{
// gets the version of the driver and prints it out

       version = 1;

printf("\nVersion of Driver = %X.0%X",((version>>8) & 0x00ff), (version&0x00ff));

} // end Voc_Get_Version

//////////////////////////////////////////////////////////////////////////////

int Voc_Init_Driver(void)
{
// intialize the driver and return the status

int status = 1;

printf("\nDriver Initialized");

return(status);

} // end Voc_Init_Driver

//////////////////////////////////////////////////////////////////////////////

int Voc_Terminate_Driver(void)
{
// terminate the driver
printf("\nDriver Terminated");
return 1;
} // end Voc_Terminate_Driver

//////////////////////////////////////////////////////////////////////////////

void Voc_Set_Port(unsigned port)
{

// sets the I/O port of the sound blaster
} // Voc_Set_Port

//////////////////////////////////////////////////////////////////////////////

void Voc_Set_Speaker(unsigned on)
{

// turns the speaker on or off
} // Voc_Set_Speaker

/////////////////////////////////////////////////////////////////////////////

int Voc_Play_Sound(unsigned char far *addr,unsigned char header_length)
{
// plays a pre-loaded VOC file

       PlaySound((char*)addr, 0, SND_ASYNC);
       current_sound_play = addr;

       return 1;
} // end Voc_Play_Sound

/////////////////////////////////////////////////////////////////////////////

int Voc_Stop_Sound(void)
{
```

```
// stops a sound that is playing
       PlaySound(NULL, 0, 0);
       return 1;

} // end Voc_Stop_Sound

/////////////////////////////////////////////////////////////////////////////

int Voc_Pause_Sound(void)
{
// pauses a sound that is playing
       Voc_Stop_Sound();
       return 1;
} // end Voc_Pause_Sound

/////////////////////////////////////////////////////////////////////////////

int Voc_Continue_Sound(void)
{
// continue a paused sound a sound that is playing
       Voc_Play_Sound(current_sound_play, 0);
       return 1;
} // end Voc_Continue_Sound

/////////////////////////////////////////////////////////////////////////////

int Voc_Break_Sound(void)
{
// break a sound loop
       Voc_Stop_Sound();
       return 1;

} // end Voc_Break_Sound

/////////////////////////////////////////////////////////////////////////////

void Voc_Set_DMA(unsigned dma)
{
} // Voc_Set_DMA

//////////////////////////////////////////////////////////////////////////////

void Voc_Set_Status_Addr(char __far *status)
{
} // Voc_Set_Status_Addr

//////////////////////////////////////////////////////////////////////////////

void Voc_Load_Driver(void)
{
} // end Voc_Load_Driver

//////////////////////////////////////////////////////////////////////////////

char far *Voc_Load_Sound(char *filename, unsigned char *header_length)
{
// loads a sound off disk into memory and points a pointer to it

       int len = strlen(filename)+1;

       char* buff = new char[len];
       strcpy(buff, filename);

       (*header_length) = 1;
       return buff;
} // end Voc_Load_Sound

//////////////////////////////////////////////////////////////////////////////

void Voc_Unload_Sound(char far *sound_ptr)
{
```

```
// delete the sound from memory

       delete sound_ptr;

} // end Voc_Unload_Sound


//////////////////////////////////////////////////////////////////////////////

void main2(void)
{

char far *sounds[4];
unsigned char lengths[4];
int done=0,sel;

Voc_Load_Driver();

Voc_Init_Driver();

Voc_Set_Port(0x220);

Voc_Set_DMA(5);

Voc_Get_Version();

Voc_Set_Status_Addr((char __far *)&ct_voice_status);

// load in sounds

sounds[0] = Voc_Load_Sound("beav.wav" , &lengths[0]);
sounds[1] = Voc_Load_Sound("ed209.wav" ,&lengths[1]);
sounds[2] = Voc_Load_Sound("term.wav" , &lengths[2]);
sounds[3] = Voc_Load_Sound("driver.wav",&lengths[3]);

Voc_Set_Speaker(1);

// main event loop, let user select a sound to play, note you can interupt
// a sound that is currenlty playing

while(!done)
     {
     printf("\n\nSound Demo Menu");
     printf("\n1 - Beavis");
     printf("\n2 - ED 209");
     printf("\n3 - Terminator");
     printf("\n4 - Exit");
     printf("\n\nSelect One ? ");
     scanf("%d",&sel);

     switch (sel)
            {
            case 1:
                  {
                  Voc_Stop_Sound();
                  Voc_Play_Sound((unsigned char *)sounds[0] , lengths[0]);
                  } break;

            case 2:
                  {
                  Voc_Stop_Sound();
                  Voc_Play_Sound((unsigned char *)sounds[1] , lengths[1]); ;
                  } break;

            case 3:
                  {
                  Voc_Stop_Sound();
                  Voc_Play_Sound((unsigned char *)sounds[2] , lengths[2]); ;
                  } break;

            case 4:
                  {
                  done = 1;
                  } break;

            default:
                   {
                   printf("\nFunction %d is not a selection.",sel);
                   } break;

            } // end switch

     } // end while

// terminate

Voc_Play_Sound((unsigned char *)sounds[3] , lengths[3]); ;

// wait for end sequence to stop, the status variable will be -1 when a sound is
// playing and 0 otherwise

while(ct_voice_status!=0) {}

Voc_Set_Speaker(0);

// unload sounds

Voc_Unload_Sound(sounds[0]);
Voc_Unload_Sound(sounds[1]);
Voc_Unload_Sound(sounds[2]);
Voc_Unload_Sound(sounds[3]);

Voc_Terminate_Driver();

} // end main
```

CHAP_11

ANTS.CPP

```
//--------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//--------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//--------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <dos.h>
//#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
//#include <graph.h>

// D E F I N E S ////////////////////////////////////////////////////////////

#define ANT_NORTH 0
#define ANT_EAST  1
#define ANT_SOUTH 2
#define ANT_WEST  3
#define NUM_ANTS  50

// S T R U C T U R E S ///////////////////////////////////////////////////////
// ant structure

typedef struct ant_typ
        {
        int x,y;                // position of ant
        int state;              // state of ant
        unsigned char color;    // color of ant, red or green
        unsigned back_color;    // background under ant

        } ant, *ant_ptr;

// G L O B A L S ////////////////////////////////////////////////////////////

unsigned char far *video_buffer = (unsigned char *)0xA0000000L; // vram byte ptr

// unsigned int far *clock = (unsigned int far *)0x0000046C; // pointer to internal
                                                          // 18.2 clicks/sec
// USAGE: ULONG now = (*clock)();
ULONG (*clock)() = PMEM_0x0000046C;

// our little ants

ant ants[NUM_ANTS];


// F U N C T I O N S ////////////////////////////////////////////////////////

void Timer(int clicks)
{
//------------------------------------------
// EACH CLICK IS APPROX. 55 MILLISECONDS.
// YOU MAY USE Sleep() ... BUT IT'S NOT A PLAIN DOS FEATURE...
/*
Sleep(clicks*55);
return;
*/
//------------------------------------------


// this function uses the internal time keeper timer i.e. the one that goes
// at 18.2 clicks/sec to to a time delay.  You can find a 32 bit value of
// this timer at 0000:046Ch

unsigned int now;

// get current time

now = (*clock)();

// wait till time has gone past current time plus the amount we eanted to
// wait.  Note each click is approx. 55 milliseconds.

while(abs((*clock)() - now) < clicks){ Sleep(5); }

} // end Timer


/////////////////////////////////////////////////////////////////////////////

void Plot_Pixel_Fast(int x,int y,unsigned char color)
{

// plots the pixel in the desired color a little quicker using binary shifting
// to accomplish the multiplications

// use the fact that 320*y = 256*y + 64*y = y<<8 + y<<6

video_buffer[((y<<8) + (y<<6)) + x] = color;

} // end Plot_Pixel_Fast
```

```
/////////////////////////////////////////////////////////////////////////////

unsigned char Read_Pixel_Fast(int x,int y)
{

// reads a pixel from the video buffer

// use the fact that 320*y = 256*y + 64*y = y<<8 + y<<6

return(video_buffer[((y<<8) + (y<<6)) + x]);

} // end Read_Pixel_Fast

////////////////////////////////////////////////////////////////////////////

void Draw_Ground(void)
{
int index;

// draw a bunch of grey rocks

for (index=0; index<200; index++)
    {
    Plot_Pixel_Fast(rand()%320,rand()%200, 7 + rand()%2);

    } // end for index

} // end Draw_Ground

////////////////////////////////////////////////////////////////////////////

void Initialize_Ants(void)
{
int index;

for (index=0; index<NUM_ANTS; index++)
    {
    // select a random position, color and state for each ant, also scan
    // their background

    ants[index].x     = rand()%320;
    ants[index].y     = rand()%200;
    ants[index].state = rand()%4;

    if (rand()%2==1)
       ants[index].color = 10;
    else
       ants[index].color = 12;

    // scan background

    ants[index].back_color = Read_Pixel_Fast(ants[index].x, ants[index].y);

    } // end for index

} // end Initialize_Ants

////////////////////////////////////////////////////////////////////////////

void Erase_Ants(void)
{
int index;

// loop through the ant array and erase all ants by replacing what was under
// them

for (index=0; index<NUM_ANTS; index++)
    {
    Plot_Pixel_Fast(ants[index].x, ants[index].y, ants[index].back_color);
    } // end for index
} // end Erase_Ants

///////////////////////////////////////////////////////////////////////////

void Move_Ants(void)
{

int index,rock;

// loop through the ant array and move each ant depending on it's state

for (index=0; index<NUM_ANTS; index++)
    {

    // what state is the ant in?

    switch(ants[index].state)
          {

          case ANT_NORTH:
               {
               ants[index].y--;

               } break;

          case ANT_SOUTH:
               {
               ants[index].y++;

               } break;

          case ANT_WEST:
               {
               ants[index].x--;

               } break;

          case ANT_EAST:
               {
               ants[index].x++;

               } break;

          } // end switch

    // test if the ant hit a screen boundary or a rock

    if (ants[index].x > 319)
        ants[index].x = 0;
    else
    if (ants[index].x <0)
        ants[index].x = 319;

    if (ants[index].y > 200)
        ants[index].y = 200;
    else
    if (ants[index].y <0)
        ants[index].y = 199;

    // now test if we hit a rock

    rock = Read_Pixel_Fast(ants[index].x, ants[index].y);

    if (rock)
       {
       // change states

       ants[index].state = rand()%4;  // select a new state

       } // end if
    } // end for index

} // end Move_Ants

////////////////////////////////////////////////////////////////////////////

void Behind_Ants(void)
{
int index;
// loop through the ant array and scan whats under them

for (index=0; index<NUM_ANTS; index++)
    {
    // read the pixel value and save it for later

    ants[index].back_color = Read_Pixel_Fast(ants[index].x, ants[index].y);

    } // end for index

} // end Behind_Ants

////////////////////////////////////////////////////////////////////////////

void Draw_Ants(void)
{
int index;
// loop through the ant array and draw all the ants blue or red depending
// on their type

for (index=0; index<NUM_ANTS; index++)
    {
    Plot_Pixel_Fast(ants[index].x, ants[index].y, ants[index].color);
    } // end for index
} // end Draw_Ants

// M A I N ////////////////////////////////////////////////////////////////////

void main2(void)
{

// 320x200x256 color mode

_setvideomode(_MRES256COLOR);
video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr

_settextposition(2,0);
printf("Hit any key to exit.");

// draw the world

Draw_Ground();

// set all the ants up

Initialize_Ants();

while(!kbhit())
     {
     // erase all the ants

     Erase_Ants();

     // move all the ants

     Move_Ants();

     // scan whats under the ant

     Behind_Ants();
     // now draw the ant

     Draw_Ants();

        _redraw_screen();

     // wait a little

     Timer(2);

     } // end while

// restore the old video mode

_setvideomode(_DEFAULTMODE);

} // end main
```

BALL.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <stdio.h>
#include <math.h>
//#include <graph.h>

// D E F I N E S /////////////////////////////////////////////////////////////

#define EARTH_GRAVITY 9.8

// G L O B A L S /////////////////////////////////////////////////////////////

// unsigned int far *clock = (unsigned int far *)0x0000046C; // pointer to internal
                                                         // 18.2 clicks/sec
// USAGE: ULONG now = (*clock)();
ULONG (*clock)() = PMEM_0x0000046C;

// F U N C T I O N S ////////////////////////////////////////////////////////

void Timer(int clicks)
{
//------------------------------------------
// EACH CLICK IS APPROX. 55 MILLISECONDS.
// YOU MAY USE Sleep() ... BUT IT'S NOT A PLAIN DOS FEATURE...
/*
Sleep(clicks*55);
return;
*/
//------------------------------------------


// this function uses the internal time keeper timer i.e. the one that goes
// at 18.2 clicks/sec to to a time delay.  You can find a 32 bit value of
// this timer at 0000:046Ch

unsigned int now;

// get current time
```

```
now = (*clock)();

// wait till time has gone past current time plus the amount we eanted to
// wait.  Note each click is approx. 55 milliseconds.

while(abs((*clock)() - now) < clicks){ Sleep(5); }

} // end Timer


// M A I N ////////////////////////////////////////////////////////////

void main2(void)
{

float ball_x   = 160,
      ball_y   = 50,
      ball_yv  = 0,
      ball_acc = EARTH_GRAVITY;

int done=0,key;

// use all MS graphics routines for a change

_setvideomode(_MRES256COLOR);

_settextposition(0,0);
printf("Q to quit, use +,- to change gravity.");

while(!done)
     {

     // has there been a keyboard press

     if (kbhit())
        {

        // test what key

        switch(getch())
              {
              case '-':
                      {
                      ball_acc-=.1;

                      } break;

              case '=':
                      {
                      ball_acc+=.1;

                      } break;

              case 'q':
                      {
                      done=1;

                      } break;

              } // end switch

        // let user know what the gravity is

        _settextposition(24,2);
        printf("Gravitational Constant = %f",ball_acc);

        } // end if keyboard hit

    // erase the ball

    _setcolor(0);
    _ellipse(_GBORDER, ball_x,ball_y,ball_x+10,ball_y+10);

    // move the ball

    ball_y+=ball_yv;

    // add acceleration to velocity

    ball_yv+=(ball_acc*.1); // the .1 is to scale it for viewing

    // test if ball has hit bottom

    if (ball_y>190)
       {
       ball_y=50;
       ball_yv=0;

       } // end if

    // draw ball

    _setcolor(1);

    _ellipse(_GBORDER, ball_x,ball_y,ball_x+10,ball_y+10);

    // wait a bit

    Timer(2);

    } // end while

// restore old videomode

_setvideomode(_DEFAULTMODE);

} // end main
```

GAS.CPP

```
//-------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//-------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//-------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <dos.h>
//#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
//#include <graph.h>

// D E F I N E S /////////////////////////////////////////////////////////////

#define NUM_ATOMS 300

// S T R U C T U R E S ///////////////////////////////////////////////////////

// atom structure
```

```
typedef struct ant_typ
        {
        int x,y;            // position of atom
        int xv,yv;          // velocity of atom

        } atom, *atom_ptr;

// G L O B A L S ///////////////////////////////////////////////////////////

unsigned char far *video_buffer = (unsigned char *)0xA0000000L; // vram byte ptr

// unsigned int far *clock = (unsigned int far *)0x0000046C; // pointer to internal
                                                          // 18.2 clicks/sec
// USAGE: ULONG now = (*clock)();
ULONG (*clock)() = PMEM_0x0000046C;

// our atoms

atom atoms[NUM_ATOMS];


// F U N C T I O N S ///////////////////////////////////////////////////////

void Timer(int clicks)
{
//-------------------------------------------
// EACH CLICK IS APPROX. 55 MILLISECONDS.
// YOU MAY USE Sleep() ... BUT IT'S NOT A PLAIN DOS FEATURE...
/*
Sleep(clicks*55);
return;
*/
//-------------------------------------------


// this function uses the internal time keeper timer i.e. the one that goes
// at 18.2 clicks/sec to to a time delay.  You can find a 32 bit value of
// this timer at 0000:046Ch

unsigned int now;

// get current time

now = (*clock)();

// wait till time has gone past current time plus the amount we eanted to
// wait.  Note each click is approx. 55 milliseconds.

while(abs((*clock)() - now) < clicks){ Sleep(5); }

} // end Timer


////////////////////////////////////////////////////////////////////////////

void Plot_Pixel_Fast(int x,int y,unsigned char color)
{

// plots the pixel in the desired color a little quicker using binary shifting
// to accomplish the multiplications

// use the fact that 320*y = 256*y + 64*y = y<<8 + y<<6

video_buffer[((y<<8) + (y<<6)) + x] = color;

} // end Plot_Pixel_Fast
```

```
////////////////////////////////////////////////////////////////////////////

void Initialize_Atoms(void)
{
int index;

for (index=0; index<NUM_ATOMS; index++)
    {
    // select a random position and trajectory for each atom
    // their background

    atoms[index].x     = 5 + rand()%300;
    atoms[index].y     = 20 + rand()%160;

    atoms[index].xv    = -5 + rand()%10;
    atoms[index].yv    = -5 + rand()%10;

    } // end for index

} // end Initialize_Atoms

////////////////////////////////////////////////////////////////////////////

void Erase_Atoms(void)
{
int index;

// loop through the atoms and erase them

for (index=0; index<NUM_ATOMS; index++)
    {
    Plot_Pixel_Fast(atoms[index].x, atoms[index].y, 0);
    } // end for index

} // end Erase_Atoms

////////////////////////////////////////////////////////////////////////////

void Move_Atoms(void)
{

int index;;

// loop through the atom array and move each atom also check collsions
// with the walls of the container

for (index=0; index<NUM_ATOMS; index++)
    {

    // move the atoms

    atoms[index].x+=atoms[index].xv;
    atoms[index].y+=atoms[index].yv;

    // did the atom hit a wall, if so reflect the velocity vector

    if (atoms[index].x > 310 || atoms[index].x <10)
        {
        atoms[index].xv=-atoms[index].xv;
        atoms[index].x+=atoms[index].xv;
        } // end if hit a vertical wall


    if (atoms[index].y > 190 || atoms[index].y <30)
       {
       atoms[index].yv=-atoms[index].yv;
       atoms[index].y+=atoms[index].yv;
       } // end if hit a horizontal wall

    } // end for index

} // end Move_Atoms

////////////////////////////////////////////////////////////////////////////

void Draw_Atoms(void)
{
int index;

// loop through the atoms and draw them

for (index=0; index<NUM_ATOMS; index++)
    {
    Plot_Pixel_Fast(atoms[index].x, atoms[index].y, 10);
    } // end for index

} // end Draw_Atoms

// M A I N ////////////////////////////////////////////////////////////////

void main2(void)
{

// 320x200x256 color mode

_setvideomode(_MRES256COLOR);
video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr

_settextposition(2,0);
printf("Hit any key to exit.");

// draw the container

_setcolor(9);

_rectangle(_GBORDER,0,16,319,199);

// set all the ants up

Initialize_Atoms();

while(!kbhit())
     {
     // erase all the atoms

     Erase_Atoms();

     // move all the atoms

     Move_Atoms();

     // now draw the atoms

     Draw_Atoms();
_redraw_screen();

     // wait a little

     Timer(1);

     } // end while

// restore the old video mode
```

```
_setvideomode(_DEFAULTMODE);

} // end main
```

CHAP_12

SPY.CPP

```
//-------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//-------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//-------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <dos.h>
//#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
//#include <graph.h>

// D E F I N E S /////////////////////////////////////////////////////////////

#define TIME_KEEPER_INT 0x1C


// G L O B A L S /////////////////////////////////////////////////////////////

void (_interrupt _far *Old_Isr)();  // holds old com port interrupt handler

long time=0;

// F U N C T I O N S /////////////////////////////////////////////////////////

void _interrupt _far Timer()
{

// increment global time variable, note: we can do this since on entry
// DS points to global data segment

time++;

} // end Timer



// M A I N ///////////////////////////////////////////////////////////////////

void main2(void)
{

// install our ISR

Old_Isr = _dos_getvect(TIME_KEEPER_INT);

_dos_setvect(TIME_KEEPER_INT, Timer);

// wait for user to hit a key

while(!kbhit())
     {
       FAST_CPU_WAIT(10);

     // print the time variable out, note: the main does NOT touch it...

     _settextposition(0,0);
     printf("\nThe timer reads:%ld   ",time);

     } // end while

// replace old ISR

_dos_setvect(TIME_KEEPER_INT, Old_Isr);


} // end main
```

PRES.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <dos.h>
//#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
//#include <graph.h>
//#include "serlib.h"

// D E F I N E S /////////////////////////////////////////////////////////////

#define TIME_KEEPER_INT 0x1C


// G L O B A L S /////////////////////////////////////////////////////////////

void (_interrupt _far *Old_Isr)();  // holds old com port interrupt handler

long time=0;

// F U N C T I O N S ////////////////////////////////////////////////////////

void _interrupt _far Timer()
{

// increment global time variable, note: we can do this since on entry
// DS points to global data segment

time++;

} // end Timer

/////////////////////////////////////////////////////////////////////////////

Plot_Responder()
{

static int first_time=1;
static long old_time;
// test if this is first time

if (first_time)
   {
   // reset first time

   first_time=0;

   old_time = time;

   } // end if first time
else
   { // not first time

   // have 5 clicks past?

   if ( (time-old_time)>=5)
      {
      old_time = time; // save new old time

      // plot the pixel

      _setcolor(rand()%16);
      _setpixel(rand()%320,rand()%200);

      } // end if

   } // end else

} // end Plot_Responder

// M A I N ///////////////////////////////////////////////////////////////////

void main2(void)
{

_setvideomode(_MRES256COLOR);

printf("Hit any key to exit...");

// install our ISR

Old_Isr = _dos_getvect(TIME_KEEPER_INT);

_dos_setvect(TIME_KEEPER_INT, Timer);

// wait for user to hit a key

while(!kbhit())
     {
             FAST_CPU_WAIT(10);

     // .. game code

     // call all responders

     Plot_Responder();

     // .. more game code

     } // end while

_setvideomode(_DEFAULTMODE);

// replace old ISR

_dos_setvect(TIME_KEEPER_INT, Old_Isr);

} // end main
```

OUTATIME.CPP

```
//-------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//-------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//-------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <stdio.h>
#include <conio.h>

// D E F I N E S ////////////////////////////////////////////////////////////

#define CONTROL_8253  0x43  // the 8253's control register
#define CONTROL_WORD  0x3C  // the control word to set mode 2,binary least/most
#define COUNTER_0     0x40  // counter 0

#define TIMER_60HZ    0x4DAE // 60 hz
#define TIMER_30HZ    0x965C // 30 hz
#define TIMER_20HZ    0xE90B // 20 hz
#define TIMER_18HZ    0xFFFF // 18.2 hz (the standard count)

// M A C R O S //////////////////////////////////////////////////////////////

#define LOW_BYTE(n) (n & 0x00ff)
#define HI_BYTE(n) ((n>>8) & 0x00ff)

// F U N C T I O N S ////////////////////////////////////////////////////////

Change_Time(unsigned int new_count)
{

// send the control word, mode 2, binary, least/most

_outp(CONTROL_8253, CONTROL_WORD);

// now write the least significant byte to the counter register

_outp(COUNTER_0,LOW_BYTE(new_count));

// and now the hi byte

_outp(COUNTER_0,HI_BYTE(new_count));


} // end Change_Time

// M A I N //////////////////////////////////////////////////////////////////

void main2(void)
{

// reprogram the timer to 60 hz instead of 18.2 hz

Change_Time(TIMER_60HZ);

} // end main
```

STARS.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <dos.h>
//#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
//#include <graph.h>


// D E F I N E S /////////////////////////////////////////////////////////////

#define TIME_KEEPER_INT 0x1C
#define NUM_STARS 50

// S T R U C T U R E S ///////////////////////////////////////////////////////

typedef struct star_typ
        {
        int x,y;    // position of star
        int vel;    // x - component of star velocity
        int color;  // color of star

        } star, *star_ptr;


// G L O B A L S /////////////////////////////////////////////////////////////

void (_interrupt _far *Old_Isr)();  // holds old com port interrupt handler

unsigned char far *video_buffer = (unsigned char far *)0xA0000000L; // vram byte ptr

int star_first=1;  // flags first time into star field

star stars[NUM_STARS]; // the star field


// F U N C T I O N S ////////////////////////////////////////////////////////

Plot_Pixel_Fast(int x,int y,unsigned char color)
{

// plots the pixel in the desired color a little quicker using binary shifting
// to accomplish the multiplications

// use the fact that 320*y = 256*y + 64*y = y<<8 + y<<6

video_buffer[((y<<8) + (y<<6)) + x] = color;

} // end Plot_Pixel_Fast

//////////////////////////////////////////////////////////////////////////////

void _interrupt _far Star_Int()
{

// this function will create a panning 3-d star field with 3-planes, like
// looking out of the Enterprise

// note: this function had better execute faster than 55.4 ms, otherwise it
// will be called again re-entrantly and kaboom!
```

```
int index;

// test if we need to initialize star field i.e. first time function is being
// called

if (star_first)
   {
   // reset first time
   star_first=0;

   // initialize all the stars

   for (index=0; index<NUM_STARS; index++)
       {
       // initialize each star to a velocity, position and color

       stars[index].x     = rand()%320;
       stars[index].y     = rand()%180;

       // decide what star plane the star is in

       switch(rand()%3)
             {
             case 0: // plane 1- the farthest star plane
                  {
                  // set velocity and color

                  stars[index].vel = 2;
                  stars[index].color = 8;

                  } break;

             case 1: // plane 2-The medium distance star plane
                  {

                  stars[index].vel = 4;
                  stars[index].color = 7;

                  } break;

             case 2: // plane 3-The nearest star plane
                  {

                  stars[index].vel = 6;
                  stars[index].color = 15;

                  } break;

             } // end switch

       } // end for index

   } // end if first time
else
   { // must be nth time in, so do the usual

   // erase, move, draw

   for (index=0; index<NUM_STARS; index++)
       {
       // erase

       Plot_Pixel_Fast(stars[index].x,stars[index].y,0);

       // move

       if ( (stars[index].x+=stars[index].vel) >=320 )
          stars[index].x = 0;

       // draw
       Plot_Pixel_Fast(stars[index].x,stars[index].y,stars[index].color);

       } // end for index

   _redraw_screen();

   } // end else

} // end Star_Int

// M A I N ///////////////////////////////////////////////////////////////////

void main2(void)
{
int num1, num2,c;

_setvideomode(_MRES256COLOR);
video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr

// install our ISR

Old_Isr = _dos_getvect(TIME_KEEPER_INT);

_dos_setvect(TIME_KEEPER_INT, Star_Int);

// wait for user to hit a key

_settextposition(23,0);

printf("Hit Q - to quit.");
printf("\nHit E - to see something wonderful...");

// get the character

c = getch();

// does user feel adventurous

if (c=='e')
   {
   printf("\nLook stars in DOS, how can this be ?");

   exit(0);  // exit without fixing up old ISR
   } // end if

// replace old ISR

_dos_setvect(TIME_KEEPER_INT, Old_Isr);

_setvideomode(_DEFAULTMODE);

} // end main
```

CYBER.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


// I N C L U D E S ///////////////////////////////////////////////////////////

#include <dos.h>
//#include <bios.h>
```

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
//#include <graph.h>

// D E F I N E S /////////////////////////////////////////////////////////////

#define KEYBOARD_INT    0x09
#define KEY_BUFFER      0x60
#define KEY_CONTROL     0x61
#define INT_CONTROL     0x20

// make and break codes for the arrow keys

#define MAKE_RIGHT      77
#define MAKE_LEFT       75
#define MAKE_UP         72
#define MAKE_DOWN       80

#define BREAK_RIGHT     205
#define BREAK_LEFT      203
#define BREAK_UP        200
#define BREAK_DOWN      208

// indices into arrow key state table

#define INDEX_UP        0
#define INDEX_DOWN      1
#define INDEX_RIGHT     2
#define INDEX_LEFT      3

// G L O B A L S /////////////////////////////////////////////////////////////

void (_interrupt _far *Old_Isr)();  // holds old com port interrupt handler

unsigned char far *video_buffer = (unsigned char far *)0xA0000000L; // vram byte ptr

int raw_key;  // the global raw keyboard data

int key_table[4] = {0,0,0,0}; // the arrow key state table

// F U N C T I O N S ////////////////////////////////////////////////////////

Plot_Pixel_Fast(int x,int y,unsigned char color)
{

// plots the pixel in the desired color a little quicker using binary shifting
// to accomplish the multiplications

// use the fact that 320*y = 256*y + 64*y = y<<8 + y<<6

       if(x < 0 || y < 0 || x >= 320 || y >= 200) return;

video_buffer[((y<<8) + (y<<6)) + x] = color;

} // end Plot_Pixel_Fast

//////////////////////////////////////////////////////////////////////////////

Fun_Back()
{
int index;
// draws a background that should jog your memory

_setcolor(1);
_rectangle(_GFILLINTERIOR, 0,0,320,200);

_setcolor(15);

for (index=0; index<10; index++)
    {
```

```
    _moveto(16+index*32,0);
    _lineto(16+index*32,199);

    } // end for index

for (index=0; index<10; index++)
    {
    _moveto(0,10+index*20);
    _lineto(319,10+index*20);

    } // end for index

} // end Fun_Back

//////////////////////////////////////////////////////////////////////////////

void _interrupt _far New_Key_Int()
{

// I'm in the mood for some inline!

//------------------------------------------
// NO ANY ASM
//------------------------------------------
/*
_asm
   {
   sti                       ; re-enable interrupts
   in al, KEY_BUFFER         ; get the key that was pressed
   xor ah,ah                 ; zero out upper 8 bits of AX
   mov raw_key, ax           ; store the key in global
   in al, KEY_CONTROL        ; set the control register
   or al, 82h                ; set the proper bits to reset the FF
   out KEY_CONTROL,al        ; send the new data back to the control register
   and al,7fh
   out KEY_CONTROL,al        ; complete the reset
   mov al,20h
   out INT_CONTROL,al        ; re-enable interrupts
                             ; when this baby hits 88 mph, your gonna see
                             ; some serious @#@#$%

   } // end inline assembly
*/
//------------------------------------------

             raw_key = (unsigned)_inp(KEY_BUFFER);

// now for some C to update the arrow state table

// process the key and update the table

switch(raw_key)
      {
      case MAKE_UP:    // pressing up
           {
           key_table[INDEX_UP]    = 1;
           } break;

      case MAKE_DOWN:  // pressing down
           {
           key_table[INDEX_DOWN]  = 1;
           } break;

      case MAKE_RIGHT: // pressing right
           {
           key_table[INDEX_RIGHT] = 1;
           } break;

      case MAKE_LEFT:  // pressing left
           {
           key_table[INDEX_LEFT]  = 1;
           } break;

      case BREAK_UP:    // releasing up
           {
           key_table[INDEX_UP]    = 0;
           } break;

      case BREAK_DOWN:  // releasing down
           {
           key_table[INDEX_DOWN]  = 0;
           } break;

      case BREAK_RIGHT: // releasing right
           {
           key_table[INDEX_RIGHT] = 0;
           } break;

      case BREAK_LEFT:  // releasing left
           {
           key_table[INDEX_LEFT]  = 0;
           } break;

      default: break;


      } // end switch

} // end New_Key_Int

// M A I N ////////////////////////////////////////////////////////////////////

void main2(void)
{
int done=0,x=160,y=100; // exit flag and dot position

// 320x200x256 color mode

_setvideomode(_MRES256COLOR);
video_buffer = (unsigned char far *)MEMORY_0xA0000000; // vram byte ptr

Fun_Back(); // light cycles anyone?

printf("Use numeric pad ARROWS keys.\nPress ESC to Exit.");

// install our ISR

Old_Isr = _dos_getvect(KEYBOARD_INT);

_dos_setvect(KEYBOARD_INT, New_Key_Int);

// main event loop

while(!done)
     {
             FAST_CPU_WAIT(10);

_settextposition(24,2);

printf("raw key=%d   ",raw_key);

     // look in the table and move the little dot

     if (key_table[INDEX_RIGHT])
         x++;

     if (key_table[INDEX_LEFT])
         x--;

     if (key_table[INDEX_UP])
         y--;
     if (key_table[INDEX_DOWN])
         y++;

     // draw the cyber dot

     Plot_Pixel_Fast(x,y,10);
       _redraw_screen();

     // this is our exit key the make code for "esc" is 1.

     if (raw_key==1)
        done=1;

     } // end while

// replace old ISR

_dos_setvect(KEYBOARD_INT, Old_Isr);

_setvideomode(_DEFAULTMODE);

} // end main
```

CHAP_13

TERM.CPP

```
//-------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//-------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//-------------------------------------------

// I N C L U D E S///////////////////////////////////////////////////////////

#include <stdio.h>
//#include <graph.h>


// G L O B A L S ////////////////////////////////////////////////////////////

// unsigned int far *clock = (unsigned int far *)0x0000046C; // pointer to internal
                                                          // 18.2 clicks/sec
// USAGE: ULONG now = (*clock)();
ULONG (*clock)() = PMEM_0x0000046C;

/////////////////////////////////////////////////////////////////////////////

void Timer(int clicks)
{
//-------------------------------------------
// EACH CLICK IS APPROX. 55 MILLISECONDS.
// YOU MAY USE Sleep() ... BUT IT'S NOT A PLAIN DOS FEATURE...
/*
Sleep(clicks*55);
return;
*/
//-------------------------------------------


// this function uses the internal time keeper timer i.e. the one that goes
// at 18.2 clicks/sec to to a time delay.  You can find a 32 bit value of
// this timer at 0000:046Ch
```

```
unsigned int now;

// get current time

now = (*clock)();

// wait till time has gone past current time plus the amount we eanted to
// wait.  Note each click is approx. 55 milliseconds.

while(abs((*clock)() - now) < clicks){ Sleep(5); }

} // end Timer


// M A I N ////////////////////////////////////////////////////////////////

void main2(void)
{

int px=160,py=100, // starting position of player
    ex=0,ey=0;     // starting position of enemy

int done=0; // exit flag

_setvideomode(_MRES256COLOR);

printf("Use U, N, J, H keys to move.      The Terminator - Q to Quit");

// main game loop

while(!done)
     {
     // erase dots

     _setcolor(0);

     _setpixel(px,py);
     _setpixel(ex,ey);

     // move player

     if (kbhit())
        {

        // which way is player moving

        switch(getch())
              {

              case 'u': // up
                      {
                  py-=2;
                      } break;

              case 'n': // down
                      {
                  py+=2;
                      } break;

              case 'j': // right
                      {
                  px+=2;
                      } break;


              case 'h': // left
                      {
                  px-=2;
                      } break;

              case 'q':
                      {
                      done=1;
                      } break;

              } // end switch

        } // end if player hit a key

     // move enemy

     // begin brain

     if (px>ex) ex++;
     if (px<ex) ex--;
     if (py>ey) ey++;
     if (py<ey) ey--;

     // end brain

     // draw dots

     _setcolor(9);
     _setpixel(px,py);

     _setcolor(12);
     _setpixel(ex,ey);

     // wait a bit

     Timer(1);

     } // end while

_setvideomode(_DEFAULTMODE);

} // end main
```

FLY.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


// I N C L U D E S////////////////////////////////////////////////////////////

#include <stdio.h>
//#include <graph.h>
#include <math.h>

// G L O B A L S ////////////////////////////////////////////////////////////

// unsigned int far *clock = (unsigned int far *)0x0000046C; // pointer to internal
                                                          // 18.2 clicks/sec
// USAGE: ULONG now = (*clock)();
ULONG (*clock)() = PMEM_0x0000046C;


// the x and y components of the patterns that will be played, I just made
// them up

int patterns_x[3][20]= { 1,1,1,1,1,2,2,-1,-2,-3,-1,0,0,1,2,2,-2,-2,-1,0,
                         0,0,1,2,3,4,5,4,3,2,1,3,3,3,3,2,1,-2,-2,-1,
                         0,-1,-2,-3,-3,-2,-2,0,0,0,0,0,0,1,0,0,0,1,0,1 };
```

```
int patterns_y[3][20] = { 0,0,0,0,-1,-1,-1,-1,-1,0,0,0,0,0,2,2,2,2,2,2,
                          1,1,1,1,1,1,2,2,2,2,2,3,3,3,3,3,0,0,0,0,
                          1,1,1,2,2,-1,-1,-1,-2,-2,-1,-1,0,0,0,1,1,1,1,1 };

//////////////////////////////////////////////////////////////////////////////

void Timer(int clicks)
{
//------------------------------------------
// EACH CLICK IS APPROX. 55 MILLISECONDS.
// YOU MAY USE Sleep() ... BUT IT'S NOT A PLAIN DOS FEATURE...
/*
Sleep(clicks*55);
return;
*/
//------------------------------------------


// this function uses the internal time keeper timer i.e. the one that goes
// at 18.2 clicks/sec to to a time delay.  You can find a 32 bit value of
// this timer at 0000:046Ch

unsigned int now;

// get current time

now = (*clock)();

// wait till time has gone past current time plus the amount we eanted to
// wait.  Note each click is approx. 55 milliseconds.

while(abs((*clock)() - now) < clicks){ Sleep(5); }

} // end Timer


// M A I N ///////////////////////////////////////////////////////////////////

void main2(void)
{

int px=160,py=100, // starting position of player
    ex=0,ey=0;     // starting position of enemy

int done=0,           // exit flag
    doing_pattern=0,  // flags if a pattern is being executed
    current_pattern,  // curent pattern 0-2 that is being done by brain
    pattern_element;  // current element of pattern being executed

_setvideomode(_MRES256COLOR);

printf("Use U, N, J, H keys to move.        The Fly - Q to Quit");

// main game loop

while(!done)
     {
     // erase dots

     _setcolor(0);

     _setpixel(px,py);
     _setpixel(ex,ey);

     // move player

     if (kbhit())
        {

        // which way is player moving

        switch(getch())
              {

              case 'u': // up
                      {
                  py-=2;
                      } break;

              case 'n': // down
                      {
                  py+=2;
                      } break;

              case 'j': // right
                      {
                  px+=2;
                      } break;


              case 'h': // left
                      {
                  px-=2;
                      } break;

              case 'q':
                      {
                      done=1;
                      } break;

              } // end switch

        } // end if player hit a key

     // move enemy

     // begin brain

     if (!doing_pattern)
        {

        if (px>ex) ex++;
        if (px<ex) ex--;
        if (py>ey) ey++;
        if (py<ey) ey--;

        // check if it's time to do a pattern i.e. is enemy within 50 pixels
        // of player


        if (sqrt(.1 + (px-ex)*(px-ex) + (py-ey)*(py-ey)) < 15)
           {
           // never ever use a SQRT in a real game!

           // get a new random pattern

           current_pattern = rand()%3;

           // set brain into pattern state

           doing_pattern = 1;

           pattern_element = 0;

           } // end if within a radius of 50

        } // end if doing a pattern
    else
```

```
       {
       // move the enemy using the next pattern element of the current pattern

       ex+=patterns_x[current_pattern][pattern_element];
       ey+=patterns_y[current_pattern][pattern_element];

       // are we done doing pattern

       if (++pattern_element==20)
           {
           pattern_element = 0;
           doing_pattern = 0;
           } // end if done doing pattern

       } // end else do pattern

     // end brain

     // draw dots

     _setcolor(9);
     _setpixel(px,py);

     _setcolor(12);
     _setpixel(ex,ey);

     // wait a bit

     Timer(1);

     } // end while

_setvideomode(_DEFAULTMODE);

} // end main
```

DFLY.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


// I N C L U D E S////////////////////////////////////////////////////////////

#include <stdio.h>
//#include <graph.h>
#include <math.h>

// G L O B A L S ////////////////////////////////////////////////////////////

// unsigned int far *clock = (unsigned int far *)0x0000046C; // pointer to internal
                                                          // 18.2 clicks/sec
// USAGE: ULONG now = (*clock)();
ULONG (*clock)() = PMEM_0x0000046C;


//////////////////////////////////////////////////////////////////////////////

void Timer(int clicks)
{
//------------------------------------------
// EACH CLICK IS APPROX. 55 MILLISECONDS.
// YOU MAY USE Sleep() ... BUT IT'S NOT A PLAIN DOS FEATURE...
```

```
/*
Sleep(clicks*55);
return;
*/
//------------------------------------------


// this function uses the internal time keeper timer i.e. the one that goes
// at 18.2 clicks/sec to to a time delay.  You can find a 32 bit value of
// this timer at 0000:046Ch

unsigned int now;

// get current time

now = (*clock)();

// wait till time has gone past current time plus the amount we eanted to
// wait.  Note each click is approx. 55 milliseconds.

while(abs((*clock)() - now) < clicks){ Sleep(5); }

} // end Timer


// M A I N ///////////////////////////////////////////////////////////////////

void main2(void)
{

int ex=160,ey=100; // starting position of fly

int curr_xv=1,curr_yv=0, // current translation factors
    clicks=0;            // times when the fly is done moving in the random
                         // direction

_setvideomode(_MRES256COLOR);

printf("  The Dumb Fly - Any key to Quit");

// main game loop

while(!kbhit())
     {
     // erase dots

     _setcolor(0);

     _setpixel(ex,ey);

     // move the fly

     // begin brain

     // are we done with this direction

     if (++clicks==20)
        {
        curr_xv = -5 + rand()%10; // -5 to +5
        curr_yv = -5 + rand()%10; // -5 to +5
        clicks=0;
        } // end if time for a new direction

     // move the fly

     ex+=curr_xv;
     ey+=curr_yv;

     // make sure fly stays on paper

     if (ex>319) ex=0;
     if (ex<0)   ex=319;
     if (ey>199) ey=0;
     if (ey<0)   ey=199;

     // end brain

     // draw fly

     _setcolor(12);
     _setpixel(ex,ey);

     // wait a bit

     Timer(1);

     } // end while

_setvideomode(_DEFAULTMODE);

} // end main
```

BFLY.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------


// I N C L U D E S////////////////////////////////////////////////////////////

#include <stdio.h>
//#include <graph.h>
#include <math.h>

// D E F I N E S ///////////////////////////////////////////////////////////

#define STATE_CHASE   1
#define STATE_RANDOM  2
#define STATE_EVADE   3
#define STATE_PATTERN 4

// G L O B A L S ////////////////////////////////////////////////////////////

// unsigned int far *clock = (unsigned int far *)0x0000046C; // pointer to internal
                                                          // 18.2 clicks/sec
// USAGE: ULONG now = (*clock)();
ULONG (*clock)() = PMEM_0x0000046C;


// the x and y components of the patterns that will be played, I just made
// them up

int patterns_x[3][20]= { 1,1,1,1,1,2,2,-1,-2,-3,-1,0,0,1,2,2,-2,-2,-1,0,
                         0,0,1,2,3,4,5,4,3,2,1,3,3,3,3,2,1,-2,-2,-1,
                         0,-1,-2,-3,-3,-2,-2,0,0,0,0,0,0,1,0,0,0,1,0,1 };



int patterns_y[3][20] = { 0,0,0,0,-1,-1,-1,-1,-1,0,0,0,0,0,2,2,2,2,2,2,
                          1,1,1,1,1,1,2,2,2,2,2,2,3,3,3,3,3,0,0,0,0,
                          1,1,1,2,2,-1,-1,-1,-2,-2,-1,-1,0,0,0,1,1,1,1,1 };


//////////////////////////////////////////////////////////////////////////////
```

```
void Timer(int clicks)
{
//------------------------------------------
// EACH CLICK IS APPROX. 55 MILLISECONDS.
// YOU MAY USE Sleep() ... BUT IT'S NOT A PLAIN DOS FEATURE...
/*
Sleep(clicks*55);
return;
*/
//------------------------------------------


// this function uses the internal time keeper timer i.e. the one that goes
// at 18.2 clicks/sec to to a time delay.  You can find a 32 bit value of
// this timer at 0000:046Ch

unsigned int now;

// get current time

now = (*clock)();

// wait till time has gone past current time plus the amount we eanted to
// wait.  Note each click is approx. 55 milliseconds.

while(abs((*clock)() - now) < clicks){ Sleep(5); }

} // end Timer


// M A I N ///////////////////////////////////////////////////////////////////

void main2(void)
{

int px=160,py=100,    // starting position of player
    ex=0,ey=0,        // starting position of enemy
    curr_xv,curr_yv;  // velocity of fly during random walk


int done=0,           // exit flag
    doing_pattern=0,  // flags if a pattern is being executed
    current_pattern,  // curent pattern 0-2 that is being done by brain
    pattern_element,  // current element of pattern being executed
    select_state=0,   // flags if a state transition needs to take place
    clicks=20,        // used to time the number of cycles a state stays active
    fly_state = STATE_CHASE;  // start fly off in chase state

float distance;       // used to hold distance between fly and player

_setvideomode(_MRES256COLOR);

printf("Use U, N, J, H keys to move.      Brainy Fly - Q to Quit");

// main game loop

while(!done)
     {
     // erase dots

     _setcolor(0);

     _setpixel(px,py);
     _setpixel(ex,ey);

     // move player

     if (kbhit())
        {
```

```
   // which way is player moving

   switch(getch())
         {

         case 'u': // up
                 {
             py-=2;
                 } break;

         case 'n': // down
                 {
             py+=2;
                 } break;

         case 'j': // right
                 {
             px+=2;
                 } break;


         case 'h': // left
                 {
             px-=2;
                 } break;

         case 'q':
                 {
                 done=1;
                 } break;

         } // end switch

   } // end if player hit a key


// move enemy

// begin brain


// what state is brain in let FSM sort it out


switch(fly_state)
      {

      case STATE_CHASE:
           {
           _settextposition(24,2);
           printf("current state:chase   ");

           // make the fly chase the player

           if (px>ex) ex++;
           if (px<ex) ex--;
           if (py>ey) ey++;
           if (py<ey) ey--;

           // time to go to another state

           if (--clicks==0)
              select_state=1;

           } break;

      case STATE_RANDOM:
           {
           _settextposition(24,2);
           printf("current state:random  ");
```

```
     // move fly in random direction

     ex+=curr_xv;
     ey+=curr_yv;

     // time to go to another state

     if (--clicks==0)
        select_state=1;

     } break;


case STATE_EVADE:
     {
     _settextposition(24,2);
     printf("current state:evade  ");


     // make fly run from player

     if (px>ex) ex--;
     if (px<ex) ex++;
     if (py>ey) ey--;
     if (py<ey) ey++;

     // time to go to another state

     if (--clicks==0)
        select_state=1;

     } break;

case STATE_PATTERN:
     {
     _settextposition(24,2);
     printf("current state:pattern  ");

     // move the enemy using the next pattern element of the current pattern

     ex+=patterns_x[current_pattern][pattern_element];
     ey+=patterns_y[current_pattern][pattern_element];

     // are we done doing pattern

     if (++pattern_element==20)
         {
         pattern_element = 0;
         select_state=1;
         } // end if done doing pattern

     } break;


default:break;

} // end switch fly state

// does brain want another state ?

if (select_state==1)
   {

   // select a state based on the envoronment and on fuzzy logic
   // uses distance from player to selct a new state

   distance =  sqrt(.5 + fabs((px-ex)*(px-ex) + (py-ey)*(py-ey)));

   if (distance > 5 && distance <15 && rand()%2==1)
      {
      // get a new random pattern
```

```
            current_pattern = rand()%3;

            // set brain into pattern state

            fly_state = STATE_PATTERN;

            pattern_element = 0;

            } // end if close to player
         else
         if (distance < 10) // too close let's run!
            {
            clicks=20;
            fly_state = STATE_EVADE;

            } // else if too close
         else
         if (distance > 25 && distance <100 && rand()%3==1)  // let's chase player
            {
            clicks=15;
            fly_state = STATE_CHASE;

            }  // end if chase player
         else
         if (distance > 30 && rand()%2==1)
            {
            clicks=10;
            fly_state = STATE_RANDOM;

            curr_xv = -5 + rand()%10; // -5 to +5
            curr_yv = -5 + rand()%10; // -5 to +5

            } // end if random
         else
            {
            clicks=5;
            fly_state = STATE_RANDOM;

            curr_xv = -5 + rand()%10; // -5 to +5
            curr_yv = -5 + rand()%10; // -5 to +5

            } // end else

         // reset need another state flag

         select_state=0;

         } // end if we need to change to another state

// make sure fly stays on paper

if (ex>319) ex=0;
if (ex<0)   ex=319;
if (ey>199) ey=0;
if (ey<0)   ey=199;

// end brain

// draw dots

_setcolor(9);
_setpixel(px,py);

_setcolor(12);
_setpixel(ex,ey);

// wait a bit

Timer(1);
```

```
     } // end while

_setvideomode(_DEFAULTMODE);

} // end main
```

CHAP_14

NLINK.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------

// I N C L U D E S ///////////////////////////////////////////////////////////

#include <dos.h>
//#include <bios.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
//#include <graph.h>

// D E F I N E S  ////////////////////////////////////////////////////////////

// registers in UART

#define SER_RBF        0    // the read buffer
#define SER_THR        0    // the write buffer
#define SER_IER        1    // the int. enable register
#define SER_IIR        2    // the int. identification register
#define SER_LCR        3    // control data config. and divisor latch
#define SER_MCR        4    // modem control reg.
#define SER_LSR        5    // line status reg.
#define SER_MSR        6    // modem status of cts, ring etc.
#define SER_DLL        0    // the low byte of baud rate divisor
#define SER_DLH        1    // the hi byte of divisor latch

// bit patterns for control registers

#define SER_BAUD_1200  96   // baud rate divisors for 1200 baud - 19200
#define SER_BAUD_2400  48
#define SER_BAUD_9600  12
#define SER_BAUD_19200  6

#define SER_GP02        8     // enable interrupt


#define COM_1           0x3F8 // base port address of port 0
#define COM_2           0x2F8 // base port address of port 1

#define SER_STOP_1      0     // 1 stop bit per character
#define SER_STOP_2      4     // 2 stop bits per character

#define SER_BITS_5      0     // send 5 bit characters
#define SER_BITS_6      1     // send 6 bit characters
#define SER_BITS_7      2     // send 7 bit characters
#define SER_BITS_8      3     // send 8 bit characters

#define SER_PARITY_NONE 0     // no parity
#define SER_PARITY_ODD  8     // odd parity
#define SER_PARITY_EVEN 24    // even parity
```

```
#define SER_DIV_LATCH_ON 128  // used to turn reg 0,1 into divisor latch

#define PIC_IMR    0x21   // pic's interrupt mask reg.
#define PIC_ICR    0x20   // pic's interupt control reg.


#define INT_SER_PORT_0    0x0C  // port 0 interrupt com 1 & 3
#define INT_SER_PORT_1    0x0B  // port 0 interrupt com 2 & 4

#define SERIAL_BUFF_SIZE 128    // current size of circulating receive buffer


// G L O B A L S ////////////////////////////////////////////////////////

void (_interrupt _far *Old_Isr)();  // holds old com port interrupt handler


char ser_buffer[SERIAL_BUFF_SIZE];  // the receive buffer

int ser_end = -1,ser_start=-1;      // indexes into receive buffer
int ser_ch, char_ready=0;           // current character and ready flag
int old_int_mask;                   // the old interrupt mask on the PIC
int open_port;                      // the currently open port
int serial_lock = 0;                // serial ISR semaphore so the buffer
                                    // isn't altered will it is being written
                                    // to by the ISR

//////////////////////////////////////////////////////////////////////////////

void _interrupt _far Serial_Isr(void)
{

// this is the ISR (Interrupt Service Routine) for the com port.  It is very
// simple.  When it gets called, it gets the next character out of the receive
// buffer register 0 and places it into the software buffer. Note: C takes care
// of all the register saving and house work.  Cool huh!

// lock out any other functions so the buffer doesn't get corrupted

serial_lock = 1;

// place character into next position in buffer

ser_ch = _inp(open_port + SER_RBF);

// wrap buffer index around

if (++ser_end > SERIAL_BUFF_SIZE-1)
    ser_end = 0;

// move character into buffer

ser_buffer[ser_end] = ser_ch;

++char_ready;

// restore PIC

_outp(PIC_ICR,0x20);

// undo lock

serial_lock = 0;

} // end Serial_Isr

//////////////////////////////////////////////////////////////////////////////

int Ready_Serial()
{
```

```
// this functions returns true if there are any characters waiting and 0 if
// the buffer is empty

return(char_ready);

} // end Ready_Serial

//////////////////////////////////////////////////////////////////////////////

int Serial_Read()
{

// this function reads a character from the circulating buffer and returns it
// to the caller

int ch;

// wait for isr to end

while(serial_lock){}

// test if there is a character(s) ready in buffer

if (ser_end != ser_start)
   {

   // wrap buffer index if needed

   if (++ser_start > SERIAL_BUFF_SIZE-1)
       ser_start = 0;

   // get the character out of buffer

   ch = ser_buffer[ser_start];

   // one less character in buffer now

   if (char_ready > 0)
       --char_ready;

   // send data back to caller

   return(ch);

   } // end if a character is in buffer
else
   // buffer was empty return a NULL i.e. 0
   return(0);

} // end Serial_read


//////////////////////////////////////////////////////////////////////////////

Serial_Write(char ch)
{

// this function writes a character to the transmit buffer, but first it
// waits for the transmit buffer to be empty.  note: it is not interrupt
// driven and it turns of interrupts while it's working

// wait for transmit buffer to be empty

while(!(_inp(open_port + SER_LSR) & 0x20)){}

// turn off interrupts for a bit

// _asm cli

// send the character
```

```
_outp(open_port + SER_THR, ch);

// turn interrupts back on

// _asm sti

} // end Serial_Write

//////////////////////////////////////////////////////////////////////////////

Open_Serial(int port_base, int baud, int configuration)
{

// this function will open up the serial port, set it's configuration, turn
// on all the little flags and bits to make interrupts happen and load the
// ISR

// save the port for other functions

open_port = port_base;

// first set the baud rate

// turn on divisor latch registers

_outp(port_base + SER_LCR, SER_DIV_LATCH_ON);

// send low and high bytes to divsor latches

_outp(port_base + SER_DLL, baud);
_outp(port_base + SER_DLH, 0);

// set the configuration for the port

_outp(port_base + SER_LCR, configuration);

// enable the interrupts

_outp(port_base + SER_MCR, SER_GP02);

_outp(port_base + SER_IER, 1);

// hold off on enabling PIC until we have the ISR installed

if (port_base == COM_1)
   {
   Old_Isr = _dos_getvect(INT_SER_PORT_0);
   _dos_setvect(INT_SER_PORT_0, Serial_Isr);
   printf("\nOpening Communications Channel Com Port #1...\n");

   }
else
   {
   Old_Isr = _dos_getvect(INT_SER_PORT_1);
   _dos_setvect(INT_SER_PORT_1, Serial_Isr);
   printf("\nOpening Communications Channel Com Port #2...\n");
   }


// enable interrupt on PIC

old_int_mask = _inp(PIC_IMR);

_outp(PIC_IMR, (port_base==COM_1) ? (old_int_mask & 0xEF) : (old_int_mask & 0xF7 ));


} // Open_Serial

//////////////////////////////////////////////////////////////////////////////
```

```
Close_Serial(int port_base)
{

// this function closes the port which entails turning off interrupts and
// restoring the old interrupt vector

// disable the interrupts

_outp(port_base + SER_MCR, 0);

_outp(port_base + SER_IER, 0);

_outp(PIC_IMR, old_int_mask );

// reset old isr handler

if (port_base == COM_1)
   {
   _dos_setvect(INT_SER_PORT_0, Old_Isr);
   printf("\nClosing Communications Channel Com Port #1.\n");
   }
else
   {
   _dos_setvect(INT_SER_PORT_1, Old_Isr);
   printf("\nClosing Communications Channel Com Port #2.\n");
   }

} // end Close_Serial

//////////////////////////////////////////////////////////////////////////////


void main2(void)
{

char ch;
int done=0;

printf("\nNull Modem Terminal Communications Program.\n\n");

// open com 1

Open_Serial(COM_1,SER_BAUD_9600,SER_PARITY_NONE | SER_BITS_8 | SER_STOP_1);

// main loop

while(!done)
     {
      FAST_CPU_WAIT(10);

     // try and get a character from local machine

     if (kbhit())
        {
        // get the character from keyboard

        ch = getch();
        printf("%c",ch);

        // send the character to other machine

        Serial_Write(ch);

        // has user pressed ESC ? if so bail.

        if (ch==27) done=1;

        // test for CR, if so add an line feed

        if (ch==13)
```

```
            {
            printf("\n");
            Serial_Write(10);
            }

        } // end if kbhit

      // try and get a character from remote

      if (ch = Serial_Read())
         printf("%c", ch);

      if (ch == 27)
         {
         printf("\nRemote Machine Closing Connection.");
         done=1;
         } // end if remote close


      }  // end while

   // close the connection and blaze

   Close_Serial(COM_1);

   } // end main
```

CHAP_17

PARAL.H

```
//
//  Paral.h - This header defines the constants and data structures
//            used in the parallax demos.
//
    #define KEYBOARD 0x09
//
// Keyboard press/release codes for the INT 9h handler
//
    #define RIGHT_ARROW_PRESSED   77
    #define RIGHT_ARROW_REL      205
    #define LEFT_ARROW_PRESSED    75
    #define LEFT_ARROW_REL       203
    #define ESC_PRESSED          129
    #define UP_ARROW_PRESSED      72
    #define UP_ARROW_REL         200
    #define DOWN_ARROW_PRESSED    80
    #define DOWN_ARROW_REL       208

    #define VIEW_WIDTH    320
    #define VIEW_HEIGHT   150
    #define MEMBLK        VIEW_WIDTH*VIEW_HEIGHT
    #define TRANSPARENT   0       // color index of see-thru pixels
    #define TOTAL_SCROLL  320

    enum {NORMAL,RLE};
    //enum {FALSE,TRUE};

    typedef struct
    {
      char manufacturer;    /* Always set to 0 */
      char version;         /* Always 5 for 256-color files */
      char encoding;        /* Always set to 1 */
      char bits_per_pixel;  /* Should be 8 for 256-color files */
      short int  xmin,ymin;       /* Coordinates for top left corner */
      short int  xmax,ymax;       /* Width and height of image */
      short int  hres;            /* Horizontal resolution of image */
      short int  vres;            /* Vertical resolution of image */
      char palette16[48];   /* EGA palette; not used for 256-color files */
      char reserved;        /* Reserved for future use */
      char color_planes;    /* Color planes */
      short int  bytes_per_line;  /* Number of bytes in 1 line of pixels */
      short int  palette_type;    /* Should be 2 for color palette */
      char filler[58];      /* Nothing but junk */
    } PcxHeader;

    typedef struct
    {
      PcxHeader hdr;
      char *bitmap;
      char pal[768];
      unsigned imagebytes,width,height;
    } PcxFile;

    #define PCX_MAX_SIZE 1640000L
    enum {PCX_OK,PCX_NOMEM,PCX_TOOBIG,PCX_NOFILE};

#ifdef __cplusplus
extern "C" {
#endif

    int ReadPcxFile(char *filename,PcxFile *pcx);
    void _interrupt NewInt9(void);
    void RestoreKeyboard(void);
    void InitKeyboard(void);
    void SetAllRgbPalette(char *pal);
    void InitVideo(void);
    void RestoreVideo(void);
    int InitBitmaps(void);
    void FreeMem(void);
    void DrawLayers(void);
    void AnimLoop(void);
    void Initialize(void);
    void CleanUp(void);
    void OpaqueBlt(char *,int,int,int);
    void TransparentBlt_2(char *,int,int,int);

#ifdef __cplusplus
  }
#endif
```

PARAL.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------

    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <time.h>
    #include <dos.h>
    #include "paral.h"

    char *MemBuf,            // pointer to memory buffer
         *BackGroundBmp,     // pointer to background bitmap data
         *VideoRam;          // pointer to VGA memory

    PcxFile pcx;             // data structure for reading PCX files

    int volatile KeyScan;    // modified by keyboard interrupt handler
    int frames=0,                // number of frames drawn
        PrevMode;                // holds original video mode

    int background;              // tracks scroll position in background bitmap

    void _interrupt (*OldInt9)(void); // ptr to BIOS keyboard handler

//
//  This routine loads a 256 color PCX file.
//
    int ReadPcxFile(char *filename,PcxFile *pcx)
    {
      long i;
      int mode=NORMAL,nbytes;
      char abyte,*p;
      FILE *f;

      f=fopen(filename,"rb");
      if(f==NULL)
        return PCX_NOFILE;
      fread(&pcx->hdr,sizeof(PcxHeader),1,f);
      pcx->width=1+pcx->hdr.xmax-pcx->hdr.xmin;
      pcx->height=1+pcx->hdr.ymax-pcx->hdr.ymin;
      pcx->imagebytes=(unsigned int)(pcx->width*pcx->height);
      if(pcx->imagebytes > PCX_MAX_SIZE)
        return PCX_TOOBIG;
      pcx->bitmap=(char*)malloc(pcx->imagebytes);
      if(pcx->bitmap == NULL)
        return PCX_NOMEM;

      p=pcx->bitmap;
      for(i=0;i<pcx->imagebytes;i++)
      {
        if(mode == NORMAL)
        {
          abyte=fgetc(f);
          if((unsigned char)abyte > 0xbf)
          {
            nbytes=abyte & 0x3f;
            abyte=fgetc(f);
            if(--nbytes > 0)
              mode=RLE;
          }
        }
        else if(--nbytes == 0)
          mode=NORMAL;
        *p++=abyte;
      }

      fseek(f,-768L,SEEK_END);      // get palette from pcx file
      fread(pcx->pal,768,1,f);
      p=pcx->pal;
      for(i=0;i<768;i++)            // bit shift palette
             //------------------------------------------
        // ALGORITHM ERROR FIXING
             //------------------------------------------
             /*
             // PALETTE WRONG,
             // p IS char BUT MUST BE unsigned char
             *p++=(*p) >>2;
             */
             //------------------------------------------

        *p++=((unsigned char)(*p)) >>2;

      fclose(f);
      return PCX_OK;                // return success
    }

//
```

```
//  This is the new int 9h handler.  This allows for smooth interactive
//  scrolling.  If the BIOS keyboard handler was not disabled holding
//  down one of the arrow keys would overflow the keyboard buffer and
//  cause a very annoying beep.
//
    void _interrupt NewInt9(void)
    {
      register char x;

      KeyScan=_inp(0x60);       // read key code from keyboard
      x=_inp(0x61);             // tell keyboard that key was processed
      _outp(0x61,(x|0x80));
      _outp(0x61,x);
      _outp(0x20,0x20);                  // send End-Of-Interrupt
      if(KeyScan == RIGHT_ARROW_REL ||  // check for keys
         KeyScan == LEFT_ARROW_REL)
         KeyScan=0;
    }

//
//  This routine restores the original BIOS keyboard interrupt handler
//
    void RestoreKeyboard(void)
    {
      _dos_setvect(KEYBOARD,OldInt9);   // restore BIOS keyboard interrupt
    }

//
//  This routine saves the original BIOS keyboard interrupt handler and
//  then installs a customer handler for this program.
//
    void InitKeyboard(void)
    {
      OldInt9=_dos_getvect(KEYBOARD);   // save BIOS keyboard interrupt
      _dos_setvect(KEYBOARD,NewInt9);   // install new int 9h handler
    }

//
//  This routine calls the video BIOS to set all the DAC registers
//  of the VGA based on the contents of pal[].
//
    void SetAllRgbPalette(char *pal)
    {
      struct SREGS s;
      union REGS r;

      segread(&s);                    // get current segment values
      s.es=FP_SEG((void far*)pal);    // point ES to pal array
      r.x.dx=FP_OFF((void far*)pal);  // get offset to pal array
      r.x.ax=0x1012;                  // BIOS func 10h sub 12h
      r.x.bx=0;                       // starting DAC register
      r.x.cx=256;                     // ending DAC register
      int86x(0x10,&r,&r,&s);          // call video BIOS
    }

//
//  This routine sets up the video mode to BIOS mode 13h.  This mode
//  is the MCGA compatible 320x200x256 mode.
//
    void InitVideo()
    {
      union REGS r;

      r.h.ah=0x0f;                 // BIOS func 0fh
      int86(0x10,&r,&r);           // call video BIOS
      PrevMode=r.h.al;             // save current video mode
      r.x.ax=0x13;                 // set video mode 13h: 320x200x256
      int86(0x10,&r,&r);           // call video BIOS
      VideoRam=MK_FP(0xa000,0);    // create a pointer to video memory
    }
```

```
//
//  This routine restores the video mode to its original state.
//
    void RestoreVideo()
    {
      union REGS r;

      r.x.ax=PrevMode;          // restore previous video mode
      int86(0x10,&r,&r);        // call video BIOS
    }

//
//  This routine loads the bitmap layers.
//
    int InitBitmaps()
    {
      int r;

      background=1;                             // initial split location

      r=ReadPcxFile("backgrnd.pcx",&pcx);       // read in background bitmap
      if(r != PCX_OK)                           // check for errors
        return FALSE;
      BackGroundBmp=pcx.bitmap;                 // save bitmap pointer
      SetAllRgbPalette(pcx.pal);                // setup VGA palette

      MemBuf=(char*)malloc(MEMBLK);                    // create system memory buffer
      if(MemBuf == NULL)                        // check for errors
        return FALSE;

      memset(MemBuf,0,MEMBLK);                  // clear buffer
      return TRUE;                              // success!
    }

//
//  This routine frees all memory allocated by the program.
//
    void FreeMem()
    {
      free(MemBuf);
      free(BackGroundBmp);
    }

//
//  This routine draws a scrolling bitmap layer where all pixels are
//  opaque.  It uses the C function memcpy() for speed.  The argument
//  ScrollSplit defines the column which splits the bitmap into two
//  halves.
//
    void OpaqueBlt(char *bmp,int StartY,int Height,int ScrollSplit)
    {
      char *dest;
      int i;

      dest=MemBuf+StartY*320;
      for(i=0;i<Height;i++)
      {
// draw the left bitmap half in the right half of the memory buffer
        memcpy(dest+ScrollSplit,bmp,VIEW_WIDTH-ScrollSplit);
// draw the right bitmap half in the left half of the memory buffer
        memcpy(dest,bmp+VIEW_WIDTH-ScrollSplit,ScrollSplit);
        bmp+=VIEW_WIDTH;
        dest+=VIEW_WIDTH;
      }
    }

//
//  This routine draws the parallax layers.  The order of the functions
//  determines the Z-ordering of the layers.
//
    void DrawLayers()
    {
      OpaqueBlt(BackGroundBmp,0,100,background);
    }

//
//  This routine handles the animation.  Note that this is the most
//  time critical section of code.  To optimize the parallax drawing
//  this routine and its children (functions called by this routine)
//  could be re-written in assembly language.  A 100% increase in
//  drawing speed would be typical.
//
    void AnimLoop()
    {
      while(KeyScan != ESC_PRESSED)     // loop until ESC key hit
      {
        switch(KeyScan)                   // process key that was hit
        {
        case RIGHT_ARROW_PRESSED:         // right arrow is down
          background-=1;                  // scroll background left 2 pixels
          if(background < 1)              // did we reach the end?
            background+=VIEW_WIDTH;       // ...then make it wrap around

          break;
        case LEFT_ARROW_PRESSED:          // left arrow is down
          background+=1;                  // scroll background right 2 pixels
          if(background > VIEW_WIDTH-1)   // did we reach the end?
            background-=VIEW_WIDTH;       // ...then make it wrap around

          break;
        default:                          // handle any other keys
          break;
        }
        DrawLayers();                    // draw parallax layer(s) in MemBuf
        memcpy(VideoRam,MemBuf,MEMBLK);  // copy MemBuf to VGA memory
        frames++;                        // track of total frames drawn

_redraw_screen();
      }
    }

//
//  This routine performs all the initialization.
//
    void Initialize()
    {
      InitVideo();            // set up mode 13h
      InitKeyboard();         // install our keyboard handler
      if(!InitBitmaps())      // read in the bitmaps
      {
        CleanUp();            // free up memory
        printf("\nError loading bitmaps\n");
        exit(1);
      }
    }

//
//  This routine performs all the necessary cleanup
//
    void CleanUp()
    {
      RestoreVideo();       // put VGA back in original state
      RestoreKeyboard();    // restore BIOS keyboard handling
      FreeMem();            // release all memory
    }

//
//  This is the main program start.  This function calls the initialization
//  routines.  Then it gets the current clock ticks, calls the animation
//  loop, and finally gets the ending clock ticks.  The clock ticks are
//  used to calculate the animation frame rate.
//
```

```
    void main2(void)
    {
printf("\n\n\n\nUse numpad ARROWS\n\n");

      clock_t begin,fini;

      Initialize();           // set video mode, load bitmaps, etc
VideoRam = (char far *)MEMORY_0xA0000000; // vram byte ptr

      begin=clock();          // get clock ticks at animation start
      AnimLoop();             // do the animation
      fini=clock();           // get clock ticks at animation end

      CleanUp();              // free mem, etc
      printf("Frames: %d\nfps: %f\n",frames,(float)CLK_TCK*frames/(fini-begin));
    }
```

PARAL-1-2.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------

    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <time.h>
    #include <dos.h>
    #include "paral.h"

    char *MemBuf,             // pointer to memory buffer
         *BackGroundBmp,      // pointer to background bitmap data
         *ForeGroundBmp,      // pointer to foreground bitmap data
         *VideoRam;           // pointer to VGA memory

    PcxFile pcx;              // data structure for reading PCX files

    int volatile KeyScan;     // modified by keyboard interrupt handler

    int frames=0,             // number of frames drawn
        PrevMode;             // holds original video mode

    int background,  // tracks scroll position in background bitmap
        foreground,  // tracks scroll position in foreground bitmap
        position;    // tracks total scroll distance

    void _interrupt (*OldInt9)(void); // ptr to BIOS keyboard handler

//
//  This routine loads a 256 color PCX file.
//
    int ReadPcxFile(char *filename,PcxFile *pcx)
    {
      long i;
      int mode=NORMAL,nbytes;
      char abyte,*p;
      FILE *f;

      f=fopen(filename,"rb");
      if(f==NULL)
        return PCX_NOFILE;
      fread(&pcx->hdr,sizeof(PcxHeader),1,f);
      pcx->width=1+pcx->hdr.xmax-pcx->hdr.xmin;
      pcx->height=1+pcx->hdr.ymax-pcx->hdr.ymin;
      pcx->imagebytes=(unsigned int)(pcx->width*pcx->height);
      if(pcx->imagebytes > PCX_MAX_SIZE)
        return PCX_TOOBIG;
      pcx->bitmap=(char*)malloc(pcx->imagebytes);
      if(pcx->bitmap == NULL)
        return PCX_NOMEM;

      p=pcx->bitmap;
      for(i=0;i<pcx->imagebytes;i++)
      {
        if(mode == NORMAL)
        {
          abyte=fgetc(f);
          if((unsigned char)abyte > 0xbf)
          {
            nbytes=abyte & 0x3f;
            abyte=fgetc(f);
            if(--nbytes > 0)
              mode=RLE;
          }
        }
        else if(--nbytes == 0)
          mode=NORMAL;
        *p++=abyte;
      }

      fseek(f,-768L,SEEK_END);      // get palette from pcx file
      fread(pcx->pal,768,1,f);
      p=pcx->pal;
      for(i=0;i<768;i++)            // bit shift palette
             //--------------------------------------------
        // ALGORITHM ERROR FIXING
             //--------------------------------------------
             /*
             // PALETTE WRONG,
             // p IS char BUT MUST BE unsigned char
             *p++=(*p) >>2;
             */
             //--------------------------------------------

        *p++=((unsigned char)(*p)) >>2;

      fclose(f);
      return PCX_OK;                // return success
    }

//
//  This is the new int 9h handler.  This allows for smooth interactive
//  scrolling.  If the BIOS keyboard handler was not disabled holding
//  down one of the arrow keys would overflow the keyboard buffer and
//  cause a very annoying beep.
//
    void _interrupt NewInt9(void)
    {
      register char x;

      KeyScan=_inp(0x60);       // read key code from keyboard
      x=_inp(0x61);             // tell keyboard that key was processed
      _outp(0x61,(x|0x80));
      _outp(0x61,x);
      _outp(0x20,0x20);                  // send End-Of-Interrupt
      if(KeyScan == RIGHT_ARROW_REL ||  // check for keys
         KeyScan == LEFT_ARROW_REL)
         KeyScan=0;
    }

//
//  This routine restores the original BIOS keyboard interrupt handler
//
    void RestoreKeyboard(void)
    {
```

```
    _dos_setvect(KEYBOARD,OldInt9);   // restore BIOS keyboard interrupt
  }

//
//  This routine saves the original BIOS keyboard interrupt handler and
//  then installs a customer handler for this program.
//
  void InitKeyboard(void)
  {
    OldInt9=_dos_getvect(KEYBOARD);   // save BIOS keyboard interrupt
    _dos_setvect(KEYBOARD,NewInt9);   // install new int 9h handler
  }

//
//  This routine calls the video BIOS to set all the DAC registers
//  of the VGA based on the contents of pal[].
//
  void SetAllRgbPalette(char *pal)
  {
    struct SREGS s;
    union REGS r;

    segread(&s);                     // get current segment values
    s.es=FP_SEG((void far*)pal);     // point ES to pal array
    r.x.dx=FP_OFF((void far*)pal);   // get offset to pal array
    r.x.ax=0x1012;                   // BIOS func 10h sub 12h
    r.x.bx=0;                        // starting DAC register
    r.x.cx=256;                      // ending DAC register
    int86x(0x10,&r,&r,&s);           // call video BIOS
  }

//
//  This routine sets up the video mode to BIOS mode 13h.  This mode
//  is the MCGA compatible 320x200x256 mode.
//
  void InitVideo()
  {
    union REGS r;

    r.h.ah=0x0f;                  // BIOS func 0fh
    int86(0x10,&r,&r);            // call video BIOS
    PrevMode=r.h.al;              // save current video mode
    r.x.ax=0x13;                  // set video mode 13h: 320x200x256
    int86(0x10,&r,&r);            // call video BIOS
    VideoRam=MK_FP(0xa000,0);     // create a pointer to video memory
  }

//
//  This routine restores the video mode to its original state.
//
  void RestoreVideo()
  {
    union REGS r;

    r.x.ax=PrevMode;             // restore previous video mode
    int86(0x10,&r,&r);           // call video BIOS
  }

//
//  This routine loads the bitmap layers.
//
  int InitBitmaps()
  {
    int r;

    background=foreground=1;                  // initial split location

    r=ReadPcxFile("backgrnd.pcx",&pcx);       // read in background bitmap
    if(r != PCX_OK)                           // check for errors
      return FALSE;
    BackGroundBmp=pcx.bitmap;                 // save bitmap pointer
```

```
      SetAllRgbPalette(pcx.pal);              // setup VGA palette

      r=ReadPcxFile("foregrnd.pcx",&pcx);     // read in foreground bitmap
      if(r != PCX_OK)                         // check for errors
        return FALSE;
      ForeGroundBmp=pcx.bitmap;               // save bitmap pointer

      MemBuf=(char *)malloc(MEMBLK);                  // create system memory buffer
      if(MemBuf == NULL)                      // check for errors
        return FALSE;

      memset(MemBuf,0,MEMBLK);                // clear buffer
      return TRUE;                            // success!
    }

//
//  This routine frees all memory allocated by the program.
//
    void FreeMem()
    {
      free(MemBuf);
      free(BackGroundBmp);
      free(ForeGroundBmp);
    }

//
//  This routine draws a scrolling bitmap layer where all pixels are
//  opaque.  It uses the C function memcpy() for speed.  The argument
//  ScrollSplit defines the column which splits the bitmap into two
//  halves.
//
    void OpaqueBlt(char *bmp,int StartY,int Height,int ScrollSplit)
    {
      char *dest;
      int i;

      dest=MemBuf+StartY*320;
      for(i=0;i<Height;i++)
      {
// draw the left bitmap half in the right half of the memory buffer
        memcpy(dest+ScrollSplit,bmp,VIEW_WIDTH-ScrollSplit);
// draw the right bitmap half in the left half of the memory buffer
        memcpy(dest,bmp+VIEW_WIDTH-ScrollSplit,ScrollSplit);
        bmp+=VIEW_WIDTH;
        dest+=VIEW_WIDTH;
      }
    }

//
//  This routine draws parallax layer while checking for transparent
//  pixels.  This routine uses a for() loop instead of memcpy() to
//  allow for the pixel checking logic.
//
    void TransparentBlt_2(char *bmp,int StartY,int Height,int ScrollSplit)
    {
      char *dest;
      unsigned char c;
      int i,j;

      dest=MemBuf+StartY*320;      // get a pointer to the memory buffer
      for(i=0;i<Height;i++)        // draw all scanlines
      {
        for(j=0;j<VIEW_WIDTH-ScrollSplit;j++)  // draw the right half
        {
          c=*(bmp+j);                     // get a bitmap pixel
          if(c == TRANSPARENT)            // is it transparent?
            continue;                     // ...yes so don't draw it
          *(dest+j+ScrollSplit)=c;        // ...no so do draw it
        }

        for(j=0;j<ScrollSplit;j++)        // draw the left half
        {
          c=*(bmp+VIEW_WIDTH-ScrollSplit+j);     // get a bitmap pixel
          if(c == TRANSPARENT)                   // is it transparent?
            continue;                            // ...yes so don't draw it
          *(dest+j)=c;                           // ...no so do draw it
        }
        dest+=VIEW_WIDTH;  // get next row of memory buffer
        bmp+=VIEW_WIDTH;   // get next row of bitmap
      }
    }

//
//  This routine draws the parallax layers.  The order of the functions
//  determines the Z-ordering of the layers.
//
    void DrawLayers()
    {
      OpaqueBlt(BackGroundBmp,0,100,background);
      TransparentBlt_2(ForeGroundBmp,50,100,foreground);
    }

//
//  This routine handles the animation.  Note that this is the most
//  time critical section of code.  To optimize the parallax drawing
//  this routine and its children (functions called by this routine)
//  could be re-written in assembly language.  A 100% increase in
//  drawing speed would be typical.
//
    void AnimLoop()
    {
      while(KeyScan != ESC_PRESSED)        // loop until ESC key hit
      {
// FAST_CPU_WAIT(1);

        switch(KeyScan)                    // process key that was hit
        {
        case RIGHT_ARROW_PRESSED:          // right arrow is down
          position--;                      // update scroll total
          if(position < 0)                 // stop scrolling if end is reached
          {
            position=0;
            break;
          }
          background-=1;                   // scroll background left 2 pixels
          if(background < 1)               // did we reach the end?
            background+=VIEW_WIDTH;        // ...then make it wrap around

          foreground-=2;                   // scroll foreground left 4 pixels
          if(foreground < 1)               // did we reach the end?
            foreground+=VIEW_WIDTH;        // ...then make it wrap around

          break;
        case LEFT_ARROW_PRESSED:           // left arrow is down
          position++;                      // updated scroll total
          if(position > TOTAL_SCROLL)      // stop scrolling if end is reached
          {
            position=TOTAL_SCROLL;
            break;
          }
          background+=1;                   // scroll background right 2 pixels
          if(background > VIEW_WIDTH-1)    // did we reach the end?
            background-=VIEW_WIDTH;        // ...then make it wrap around

          foreground+=2;                   // scroll foreground right 4 pixels
          if(foreground > VIEW_WIDTH-1)    // did we reach the end?
            foreground-=VIEW_WIDTH;        // ...then make it wrap around

          break;
        default:                           // handle any other keys
          break;
        }
        DrawLayers();                    // draw parallax layer(s) in MemBuf
        memcpy(VideoRam,MemBuf,MEMBLK);  // copy MemBuf to VGA memory

_redraw_screen();

        frames++;                        // track of total frames drawn
      }
    }

//
//  This routine performs all the initialization.
//
    void Initialize()
    {
      position=0;
      InitVideo();            // set up mode 13h
      InitKeyboard();         // install our keyboard handler
      if(!InitBitmaps())      // read in the bitmaps
      {
        CleanUp();            // free up memory
        printf("\nError loading bitmaps\n");
        exit(1);
      }
    }

//
//  This routine performs all the necessary cleanup
//
    void CleanUp()
    {
      RestoreVideo();       // put VGA back in original state
      RestoreKeyboard();    // restore BIOS keyboard handling
      FreeMem();            // release all memory
    }

//
//  This is the main program start.  This function calls the initialization
//  routines.  Then it gets the current clock ticks, calls the animation
//  loop, and finally gets the ending clock ticks.  The clock ticks are
//  used to calculate the animation frame rate.
//
    void main2(void)
    {
printf("\n\n\n\nUse numpad ARROWS\n\n");

      clock_t begin,fini;

      Initialize();           // set video mode, load bitmaps, etc
VideoRam = (char far *)MEMORY_0xA0000000; // vram byte ptr

      begin=clock();          // get clock ticks at animation start
      AnimLoop();             // do the animation
      fini=clock();           // get clock ticks at animation end

      CleanUp();              // free mem, etc
      printf("Frames: %d\nfps: %f\n",frames,(float)CLK_TCK*frames/(fini-begin));
    }
```

TILES.H

```
//
// Tiles.h - This header includes definitions for scrolling tiled images
//

    #define NUM_TILES     17      // number of tile files
    #define TILE_WIDTH    16      // width in pixels of a tile
    #define TILE_HEIGHT   16      // height in pixels of a tile
    #define TILE_COLS     40      // width of tile map
    #define TILE_ROWS      6      // height of tile map
    #define TILES_TOTAL   (TILE_COLS*TILE_ROWS)
    #define TILES_PER_ROW (VIEW_WIDTH/TILE_WIDTH)

    #define SHIFT 4

#ifdef __cplusplus
extern "C" {
#endif

    void ReadTiles(void);
    void FreeTiles(void);
    void ReadTileMap(char *);
    void DrawTile(char *,int,int,int,int);
    void DrawTiles(int,int);

#ifdef __cplusplus
  }
#endif
```

TILES.CPP

```
//------------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//------------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//------------------------------------------

    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <time.h>
    #include <dos.h>
    #include "paral.h"
    #include "tiles.h"

    char *MemBuf,             // pointer to memory buffer
         *BackGroundBmp,      // pointer to background bitmap data
         *ForeGroundBmp,      // pointer to foreground bitmap data
         *VideoRam;           // pointer to VGA memory

    PcxFile pcx;              // data structure for reading PCX files

    int volatile KeyScan;     // modified by keyboard interrupt handler

    int frames=0,             // number of frames drawn
        PrevMode;             // holds original video mode

    int background,  // tracks scroll position in background bitmap
        foreground,  // tracks scroll position in foreground bitmap
        position;    // tracks tile scroll position

    char *tiles[NUM_TILES+1];
    int tilemap[TILES_TOTAL];

    void _interrupt (*OldInt9)(void); // ptr to BIOS keyboard handler

//
//  This routine loads a 256 color PCX file.
//
    int ReadPcxFile(char *filename,PcxFile *pcx)
    {
      long i;
      int mode=NORMAL,nbytes;
      char abyte,*p;
      FILE *f;

      f=fopen(filename,"rb");
      if(f==NULL)
        return PCX_NOFILE;
      fread(&pcx->hdr,sizeof(PcxHeader),1,f);
      pcx->width=1+pcx->hdr.xmax-pcx->hdr.xmin;
      pcx->height=1+pcx->hdr.ymax-pcx->hdr.ymin;
      pcx->imagebytes=(unsigned int)(pcx->width*pcx->height);
      if(pcx->imagebytes > PCX_MAX_SIZE)
        return PCX_TOOBIG;
      pcx->bitmap=(char*)malloc(pcx->imagebytes);
      if(pcx->bitmap == NULL)
        return PCX_NOMEM;

      p=pcx->bitmap;
      for(i=0;i<pcx->imagebytes;i++)
      {
        if(mode == NORMAL)
        {
          abyte=fgetc(f);
          if((unsigned char)abyte > 0xbf)
          {
            nbytes=abyte & 0x3f;
            abyte=fgetc(f);
            if(--nbytes > 0)
              mode=RLE;
          }
        }
        else if(--nbytes == 0)
          mode=NORMAL;
        *p++=abyte;
      }

      fseek(f,-768L,SEEK_END);      // get palette from pcx file
      fread(pcx->pal,768,1,f);
      p=pcx->pal;
      for(i=0;i<768;i++)            // bit shift palette
             //------------------------------------------
        // ALGORITHM ERROR FIXING
             //------------------------------------------
             /*
             // PALETTE WRONG,
             // p IS char BUT MUST BE unsigned char
             *p++=(*p) >>2;
             */
             //------------------------------------------

        *p++=((unsigned char)(*p)) >>2;

      fclose(f);
      return PCX_OK;                // return success
    }

//
//  This is the new int 9h handler.  This allows for smooth interactive
//  scrolling.  If the BIOS keyboard handler was not disabled holding
//  down one of the arrow keys would overflow the keyboard buffer and
//  cause a very annoying beep.
//
    void _interrupt NewInt9(void)
    {
      register char x;

      KeyScan=_inp(0x60);       // read key code from keyboard
      x=_inp(0x61);             // tell keyboard that key was processed
      _outp(0x61,(x|0x80));
      _outp(0x61,x);
      _outp(0x20,0x20);                  // send End-Of-Interrupt
      if(KeyScan == RIGHT_ARROW_REL ||  // check for keys
         KeyScan == LEFT_ARROW_REL)
         KeyScan=0;
    }
```

```
//
//  This routine restores the original BIOS keyboard interrupt handler
//
    void RestoreKeyboard(void)
    {
      _dos_setvect(KEYBOARD,OldInt9);   // restore BIOS keyboard interrupt
    }

//
//  This routine saves the original BIOS keyboard interrupt handler and
//  then installs a customer handler for this program.
//
    void InitKeyboard(void)
    {
      OldInt9=_dos_getvect(KEYBOARD);   // save BIOS keyboard interrupt
      _dos_setvect(KEYBOARD,NewInt9);   // install new int 9h handler
    }

//
//  This routine calls the video BIOS to set all the DAC registers
//  of the VGA based on the contents of pal[].
//
    void SetAllRgbPalette(char *pal)
    {
      struct SREGS s;
      union REGS r;

      segread(&s);                    // get current segment values
      s.es=FP_SEG((void far*)pal);    // point ES to pal array
      r.x.dx=FP_OFF((void far*)pal);  // get offset to pal array
      r.x.ax=0x1012;                  // BIOS func 10h sub 12h
      r.x.bx=0;                       // starting DAC register
      r.x.cx=256;                     // ending DAC register
      int86x(0x10,&r,&r,&s);          // call video BIOS
    }

//
//  This routine sets up the video mode to BIOS mode 13h.  This mode
//  is the MCGA compatible 320x200x256 mode.
//
    void InitVideo()
    {
      union REGS r;

      r.h.ah=0x0f;               // BIOS func 0fh
      int86(0x10,&r,&r);         // call video BIOS
      PrevMode=r.h.al;           // save current video mode
      r.x.ax=0x13;               // set video mode 13h: 320x200x256
      int86(0x10,&r,&r);         // call video BIOS
      VideoRam=MK_FP(0xa000,0);  // create a pointer to video memory
    }

//
//  This routine restores the video mode to its original state.
//
    void RestoreVideo()
    {
      union REGS r;

      r.x.ax=PrevMode;          // restore previous video mode
      int86(0x10,&r,&r);        // call video BIOS
    }

//
//  This routine loads the bitmap layers.
//
    int InitBitmaps()
    {
      int r;

      background=foreground=1;                // initial split location
      r=ReadPcxFile("backgrnd.pcx",&pcx);     // read in background bitmap
      if(r != PCX_OK)                         // check for errors
        return FALSE;
      BackGroundBmp=pcx.bitmap;               // save bitmap pointer
      SetAllRgbPalette(pcx.pal);              // setup VGA palette

      r=ReadPcxFile("foregrnd.pcx",&pcx);     // read in foreground bitmap
      if(r != PCX_OK)                         // check for errors
        return FALSE;
      ForeGroundBmp=pcx.bitmap;               // save bitmap pointer

      MemBuf=(char *)malloc(MEMBLK);                  // create system memory buffer
      if(MemBuf == NULL)                      // check for errors
        return FALSE;

      memset(MemBuf,0,MEMBLK);                // clear buffer
      return TRUE;                            // success!
    }

//
//  This routine frees all memory allocated by the program.
//
    void FreeMem()
    {
      free(MemBuf);
      free(BackGroundBmp);
      free(ForeGroundBmp);
      FreeTiles();
    }

//
//  This routine draws a scrolling bitmap layer where all pixels are
//  opaque.  It uses the C function memcpy() for speed.  The argument
//  ScrollSplit defines the column which splits the bitmap into two
//  halves.
//
    void OpaqueBlt(char *bmp,int StartY,int Height,int ScrollSplit)
    {
      char *dest;
      int i;

      dest=MemBuf+StartY*320;
      for(i=0;i<Height;i++)
      {
// draw the left bitmap half in the right half of the memory buffer
        memcpy(dest+ScrollSplit,bmp,VIEW_WIDTH-ScrollSplit);
// draw the right bitmap half in the left half of the memory buffer
        memcpy(dest,bmp+VIEW_WIDTH-ScrollSplit,ScrollSplit);
        bmp+=VIEW_WIDTH;
        dest+=VIEW_WIDTH;
      }
    }

//
//  This routine draws parallax layer while checking for transparent
//  pixels.  This routine uses a for() loop instead of memcpy() to
//  allow for the pixel checking logic.
//
    void TransparentBlt_2(char *bmp,int StartY,int Height,int ScrollSplit)
    {
      char *dest;
      unsigned char c;
      int i,j;

      dest=MemBuf+StartY*320;      // get a pointer to the memory buffer
      for(i=0;i<Height;i++)        // draw all scanlines
      {
        for(j=0;j<VIEW_WIDTH-ScrollSplit;j++)  // draw the right half
        {
          c=*(bmp+j);                     // get a bitmap pixel
          if(c == TRANSPARENT)            // is it transparent?
            continue;                     // ...yes so don't draw it
          *(dest+j+ScrollSplit)=c;        // ...no so do draw it
        }

        for(j=0;j<ScrollSplit;j++)        // draw the left half
        {
          c=*(bmp+VIEW_WIDTH-ScrollSplit+j);   // get a bitmap pixel
          if(c == TRANSPARENT)                 // is it transparent?
            continue;                          // ...yes so don't draw it
          *(dest+j)=c;                         // ...no so do draw it
        }
        dest+=VIEW_WIDTH;  // get next row of memory buffer
        bmp+=VIEW_WIDTH;   // get next row of bitmap
      }
    }

//
//  This routine draws the parallax layers.  The order of the functions
//  determines the Z-ordering of the layers.
//
    void DrawLayers()
    {
      OpaqueBlt(BackGroundBmp,0,100,background);
      TransparentBlt_2(ForeGroundBmp,50,100,foreground);
      DrawTiles(position,54);
    }

//
//  This routine handles the animation.  Note that this is the most
//  time critical section of code.  To optimize the parallax drawing
//  this routine and its children (functions called by this routine)
//  could be re-written in assembly language.  A 100% increase in
//  drawing speed would be typical.
//
    void AnimLoop()
    {
      while(KeyScan != ESC_PRESSED)       // loop until ESC key hit
      {
        switch(KeyScan)                   // process key that was hit
        {
        case RIGHT_ARROW_PRESSED:         // right arrow is down
          position+=4;                    // update tile scroll total
          if(position > TOTAL_SCROLL)     // stop scrolling if end is reached
          {
            position=TOTAL_SCROLL;
            break;
          }

          background-=1;                  // scroll background left 2 pixels
          if(background < 1)              // did we reach the end?
            background+=VIEW_WIDTH;       // ...then make it wrap around

          foreground-=2;                  // scroll foreground left 4 pixels
          if(foreground < 1)              // did we reach the end?
            foreground+=VIEW_WIDTH;       // ...then make it wrap around

          break;
        case LEFT_ARROW_PRESSED:          // left arrow is down
          position-=4;                    // update tile scroll total
          if(position < 0)                // stop scrolling if end is reached
          {
            position=0;
            break;
          }

          background+=1;                  // scroll background right 2 pixels
          if(background > VIEW_WIDTH-1)   // did we reach the end?
            background-=VIEW_WIDTH;       // ...then make it wrap around

          foreground+=2;                  // scroll foreground right 4 pixels
          if(foreground > VIEW_WIDTH-1)   // did we reach the end?
            foreground-=VIEW_WIDTH;       // ...then make it wrap around

          break;
        default:                          // handle any other keys
          break;
        }
        DrawLayers();                    // draw parallax layer(s) in MemBuf
        memcpy(VideoRam,MemBuf,MEMBLK);  // copy MemBuf to VGA memory
        frames++;                        // track of total frames drawn

_redraw_screen();
      }
    }

//
//  This routine performs all the initialization.
//
    void Initialize()
    {
      position=0;
      InitVideo();            // set up mode 13h
      InitKeyboard();         // install our keyboard handler
      if(!InitBitmaps())      // read in the bitmaps
      {
        CleanUp();            // free up memory
        printf("\nError loading bitmaps\n");
        exit(1);
      }
      ReadTileMap("tilemap.dat");
      ReadTiles();
    }

//
//  This routine performs all the necessary cleanup
//
    void CleanUp()
    {
      RestoreVideo();       // put VGA back in original state
      RestoreKeyboard();    // restore BIOS keyboard handling
      FreeMem();            // release all memory
    }

    void ReadTiles(void)
    {
      PcxFile pcx;
      char buf[80];
      int i,result;

      tiles[0]=NULL;              // setup empty tile
      for(i=1;i<=NUM_TILES;i++)
      {
        sprintf(buf,"t%d.pcx",i);
        result=ReadPcxFile(buf,&pcx);
        if(result != PCX_OK)
        {
          printf("\nerror reading file: %s\n",buf);
          exit(1);
        }
        tiles[i]=pcx.bitmap;
      }
    }

    void FreeTiles()
    {
      int i;

      for(i=0;i<NUM_TILES;i++)
        free(tiles[i]);
    }
    void ReadTileMap(char *filename)
    {
      int i;
      FILE *f;

      f=fopen(filename,"rt");
      for(i=0;i<TILES_TOTAL;i++)
      {
        fscanf(f,"%d",&(tilemap[i]));
      }
      fclose(f);
    }

//
//  This routine draws a bitmap tile in a memory buffer.  The routine
//  can draw portions of the tile smaller.  The argument 'offset' defines
//  the starting column within the tile.  The argument 'width' defines
//  the length of the tile to draw.
//
    void DrawTile(char *bmp,int x,int y,int offset,int width)
    {
      char *dest;
      int i;

      if(bmp == NULL) return;        // don't draw empty tiles
      dest=MemBuf+y*VIEW_WIDTH+x;    // calc dest offset into memory buf
      bmp+=offset;                   // get start of bitmap plus offset
      for(i=0;i<TILE_HEIGHT;i++)     // draw each scanline of bitmap
      {
        memcpy(dest,bmp,width);      // copy width bytes from bitmap into MemBuf
        dest+=VIEW_WIDTH;            // get next row of MemBuf
        bmp+=TILE_WIDTH;             // get next row of bitmap
      }
    }

//
//  This routine draws a screen full of tiles.  The argument 'vloc'
//  is the left corner x location within the virtual screen.
//
    void DrawTiles(int VirtualX,int Starty)
    {
      int i,x,index,offset,row,limit;

      index=VirtualX>>SHIFT;     // get index of first visible tile
      offset=VirtualX-(index<<SHIFT);
      limit=TILES_PER_ROW;
      if(offset==0)
        limit--;
      for(row=Starty;row<Starty+TILE_HEIGHT*TILE_ROWS;row+=TILE_HEIGHT)
      {
        x=TILE_WIDTH-offset;
// draw the leftmost tile of the current row.  May be a partial tile
        DrawTile(tiles[tilemap[index]],0,row,offset,TILE_WIDTH-offset);
        for(i=index+1;i<index+limit;i++)
        {
// draw the next tile on the current row.  Always a full tile.
          DrawTile(tiles[tilemap[i]],x,row,0,TILE_WIDTH);
          x+=TILE_WIDTH;
        }
// draw the rightmost tile of the current row.  May be a partial tile.
        DrawTile(tiles[tilemap[i]],x,row,0,offset);
        index+=TILE_COLS;
      }
    }

//
//  This is the main program start.  This function calls the initialization
//  routines.  Then it gets the current clock ticks, calls the animation
//  loop, and finally gets the ending clock ticks.  The clock ticks are
//  used to calculate the animation frame rate.
//
```

```
    void main2(void)
    {
printf("\n\n\n\nUse numpad ARROWS\n\n");

      clock_t begin,fini;

      Initialize();          // set video mode, load bitmaps, etc
VideoRam = (char far *)MEMORY_0xA0000000; // vram byte ptr

      begin=clock();         // get clock ticks at animation start
      AnimLoop();            // do the animation
      fini=clock();          // get clock ticks at animation end

      CleanUp();             // free mem, etc
      printf("Frames: %d\nfps: %f\n",frames,(float)CLK_TCK*frames/(fini-begin));
    }
```

CHAP_18

LOOKNUP.CPP

```
//-----------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//-----------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//-----------------------------------------


#include <math.h>
#include <stdio.h>
//#include <graph.h>


float sin_table[360], cos_table[360];

void main2(void)
{

int index,x,y,xo,yo,radius,color,ang;

char far *screen = (char far *)0xA0000000;

// use Microsofts library to go into 320x200x256 mode
_setvideomode(_MRES256COLOR);
screen = (char far *)MEMORY_0xA0000000; // vram byte ptr

// create look up tables

int MAX_SCREEN_OFFSET = 320*200;

for (index=0; index<360; index++)
     {
     sin_table[index]= sin(index*3.14159/180);
     cos_table[index] = cos(index*3.14159/180);

     } // end for index

// draw 1000 circles using built in sin and cos

for (index=0; index<1000; index++)
     {
     // get a random circle
     radius = rand()%50;
     xo = rand()%320;
     yo = rand()%200;
     color = rand()%256;
     for (ang=0; ang<360; ang++)
          {
          x = xo + cos(ang*3.14159/180) * radius;
          y = yo + sin(ang*3.14159/180)*radius;
          // plot the point of the circle

               int offset = (y<<6) + (y<<8) + x;
               // ALGORITHM ERROR FIXING. CHECK THE BOUNDS!
               if(offset < 0 || offset >= MAX_SCREEN_OFFSET) continue;

          screen[offset] = color;
          } // end for ang
_redraw_screen();
     } // end for index

// done, halt the system and wait for user to hit a key
printf("\nHit a key to see circles drawn twith look up tables.");

getch();
_setvideomode(_MRES256COLOR);
screen = (char far *)MEMORY_0xA0000000; // vram byte ptr

// draw 1000 circles using look up tables

for (index=0; index<1000; index++)
     {
     // get a random circle
     radius = rand()%50;
     xo = rand()%320;
     yo = rand()%200;
     color = rand()%256;

     for (ang=0; ang<360; ang++)
          {
          x = xo + cos_table[ang] * radius;
          y = yo + sin_table[ang] *radius;
          // plot the point of the circle

               int offset = (y<<6) + (y<<8) + x;
               // ALGORITHM ERROR FIXING. CHECK THE BOUNDS!
               if(offset < 0 || offset >= MAX_SCREEN_OFFSET) continue;

          screen[offset] = color;
          } // end for ang
_redraw_screen();
     } // end for index

// let user hit a key to exit

printf("\nHit any key to exit.");
getch();

_setvideomode(_DEFAULTMODE);

} // end main
```

FIX.CPP

```
//-----------------------------------------
// DOS DEVELOPMENT ENVIRONMENT EMULATION TOOLKIT
//-----------------------------------------
#include "stdafx.h"
#include "DOSEmu.h"
//-----------------------------------------
```

```
/// I N C L U D E S ///////////////////////////////////////////////////////

#include <math.h>
#include <stdio.h>

// define our new magical fixed point data type

typedef long fixed;

// F U N C T I O N S //////////////////////////////////////////////////////

fixed Assign_Integer(long integer)
{

return((fixed)integer << 8);


} // end Assign_Integer

///////////////////////////////////////////////////////////////////////////////

fixed Assign_Float(float number)
{

return((fixed)(number * 256));

} // end Assign_Float

///////////////////////////////////////////////////////////////////////////////

fixed Mul_Fixed(fixed f1,fixed f2)
{

return((f1*f2) >> 8);

} // end Mul_Fixed

///////////////////////////////////////////////////////////////////////////////

fixed Add_Fixed(fixed f1,fixed f2)
{

return(f1+f2);

} // end Add_Fixed

///////////////////////////////////////////////////////////////////////////////

Print_Fixed(fixed f1)
{

printf("%ld.%ld",f1 >> 8, 100*(unsigned long)(f1 & 0x00ff)/256);

} // end Print_Fixed


//M A I N /////////////////////////////////////////////////////////////////////


void main2(void)
{


fixed f1,f2,f3;

f1 = Assign_Float(15);
f2 = Assign_Float(233.45);

f3 = Mul_Fixed(f1,f2);

printf("\nf1:");
Print_Fixed(f1);

printf("\nf2:");
Print_Fixed(f2);

printf("\nf3:");
Print_Fixed(f3);


} // end main
```

HLINEF.CPP

```
//////////////////////////////////////////////////////////////////////////////

H_Line_Fast(int x1,int x2,int y,unsigned int color)
{
// a fast horizontal line renderer uses word writes instead of byte writes
// the only problem is the endpoints of the h line must be taken into account.
// test if the endpoints of the horizontal line are on word boundaries i.e.
// they are envenly divisible by 2
// basically, we must consider the two end points of the line separately
// if we want to write words at a time or in other words two pixels at a time
// note x2 > x1

unsigned int first_word,
            middle_word,
              last_word,
            line_offset,
                   index;


// test the 1's bit of the starting x

if ( (x1 & 0x0001) )
   {

   first_word = (color<<8);

   } // end if starting point is on a word boundary
else
   {
   // replicate color in to both bytes
   first_word = ((color<<8) | color);

   } // end else

// test the 1's bit of the ending x

if ( (x2 & 0x0001) )
   {

   last_word = ((color<<8) | color);

   } // end if ending point is on a word boundary
else
   {
   // place color in high byte of word only

   last_word = color;

   } // end else

// now we can draw the horizontal line two pixels at a time

line_offset = ((y<<7) + (y<<5));  // y*160, since there are 160 words/line

// compute middle color
middle_word = ((color<<8) | color);

// left endpoint

video_buffer_w[line_offset + (x1>>1)] = first_word;

// the middle of the line

for (index=(x1>>1)+1; index<(x2>>1); index++)
    video_buffer_w[line_offset+index] = middle_word;

// right endpoint

video_buffer_w[line_offset + (x2>>1)] = last_word;

} // end H_Line_Fast

///////////////////////////////////////////////////////////////////////////////
```