



Computer Science

Daniel Lindsäth and Martin Persson

**Implementation of a
2D Game Engine Using DirectX 8.1**

Bachelor's Project

2004:24

Implementation of a 2D Game Engine Using DirectX 8.1

Daniel Lindsäth and Martin Persson

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Daniel Lindsäth

Martin Persson

Approved, 2004-06-03

Advisor: Hannes Persson

Examiner: Donald F. Ross

Abstract

This paper describes our game engine written in C++, using the DirectX libraries for graphics, sound and input. Since the engine is written using DirectX, an introduction to this system is given. The report gives a description of the structure of the game and the game kernel. Following this is a description of the graphics engine and its core components. The main focus of the engine is on the physics and how it is used in the game to simulate reality. Input is discussed briefly, with examples to show how it relates to the physics engine. Implementation of audio in the game engine is not described, but a general description of how sound is used in games is given. A theory for the basics of how artificial intelligence can be used in the engine is presented. The system for the architecture of the levels is described as is its connection to the graphics engine. The last section of the report is an evaluation and suggestions for what to do in the future. A user manual for the level editor is included as an appendix.

Contents

1	Introduction	1
1.1	Problem	1
1.2	Purpose	2
1.3	Limitation	2
1.4	Goal	2
1.5	Disposition	3
2	DirectX and COM	4
2.1	The DirectX Kernel	4
2.2	HAL and HEL	6
2.3	The Components of DirectX	6
2.3.1	DirectDraw	6
2.3.2	DirectSound	8
2.3.3	DirectSound3D	8
2.3.4	DirectMusic	8
2.3.5	DirectInput	8
2.3.6	DirectPlay	9
2.3.7	Direct3D	9
2.3.8	DirectSetup	9
2.3.9	DirectX Graphics	9
2.3.10	DirectX Audio	10
2.3.11	DirectShow	10
2.4	COM	10
2.4.1	The COM Objects	11
2.4.2	GUID	11
2.5	DirectX and COM	12

3	Game Engine Structure	13
4	The Kernel	16
5	Graphics Engine	18
5.1	Bitmaps	18
5.2	Bitmap Object Blitter	19
6	Physics	22
6.1	Newtonian Mechanics	22
6.1.1	Movements — Acceleration and Inertia	22
6.1.2	Drag	23
6.1.3	Free-fall	24
6.2	Physics Within the Game	25
6.3	Collision Detect	27
6.3.1	Running Through Walls	27
6.3.2	The Bresenham Algorithm	29
7	Input	32
7.1	Forces	32
7.2	Impulses	33
8	Sound and Music	34
8.1	Sound in the Engine	34
9	The Basics of Artificial Intelligence	35
10	The Level System	36
11	Conclusion	38
11.1	Achievements	38

11.2 Evaluation	39
11.3 Future Plans	40
References	42
A User Manual for the Map Editor	43
B winconsole.cpp	45
C motor.h	48
D bob.h	52
E game.h	55
E.1 Terrain	56
E.2 Character	56
E.3 Enemy	57
E.4 Player	58
E.5 Sublevel	58
E.6 Level	59
E.7 Game	59
F mapeditor.h	61
G Screenshot	64

List of Figures

2.1	“Classical” Programming vs. DirectX	5
2.2	DirectX and Windows	7
3.1	Module Overview	13
3.2	Object Diagram of the Game Structure	14
4.1	Game Kernel structure	16
5.1	Bitmap Example, courtesy of [6]	18
5.2	BOB Inheritance	21
6.1	Velocity Vector	26
6.2	Collision Example	28
6.3	Another Collision Example	28
6.4	Line Approximations	30
10.1	The Different Layers	37
A.1	Empty Level Editor	43
G.1	A Screen Shot of an Example Game	64

1 Introduction

According to the authors, the gaming industry of today focuses solely on 3D games, often with wondrous graphics as their main goal. This has had the effect that modern games seldom have any depth, story or “feel” to them. The authors think that this is a shame, and decided to create a 2D game engine in the hopes of returning some of the feel of the old games.

1.1 Problem

A 2D game engine is a relatively complex thing to design. The main problem is to make the physics in the game correspond — in a good looking way — to the objects on the screen, using the existing graphics engine BOB [1]. The engine will, except for the the graphics API, be built from scratch.

Problems that have to be solved is how to make the architecture of the levels¹ be useful to the physics. A great part of the problem with the physics will be collision detection between the player and the map² itself. The input handling using DirectInput will have to be integrated to the movement physics of the player. How to make the enemies in the game orientate themselves in the environment will also have to be solved. A level editor using the same map system as the game will also have to be implemented so that levels created using that editor can be loaded into the game. The main part of the work will be programming the engine itself, and parallel to the implementation write a documentation describing the engine.

¹A level is an object containing a set of terrain, a number of enemies and events which one may interact with.

²The terrain parts of a level. Sometimes used interchangeably with level.

1.2 Purpose

The purpose of this dissertation is to create a fully working game engine that can be used to create two dimensional platform games using DirectX. The engine will also be the base that Martin will use when he creates his first commercial computer game.

1.3 Limitation

A fully functioning game engine is, of course, a massive project that can hardly be completed in the amount of time available for this paper. The authors have therefore decided to create the level system and the game physics and then make the rest of the details if there is enough time.

1.4 Goal

When this dissertation is complete the following shall be done:

- A well documented and working game engine API that allows programmers to create 2D games with a minimum amount of foundation coding, so that they may focus on the gaming experience.
- A good implementation design that will make it easy to add new features later.
- A working two dimensional physics engine.
- A useful level editor that can be used to create the maps of the game.
- A simple and general way to program the AI of the enemies in the game.
- A virtual class for enemies.

1.5 Disposition

Section 2 gives a shallow, technical introduction to DirectX and COM. These tools are then used in sections 5, 7 and 8. Section 3 shows how the modules of the game are connected and gives an introduction to sections 4 through 9.

Section 4 gives a description of the game kernel which handles startup, execution and shutdown of the game engine itself.

Section 5 explains image formats, and relates back to the second section with DirectDraw.

Sections 6 and 7 show how a player may alter the playing realm through input and physics. Section 6 also describes some of the issues related to physics in games.

Sections 8 and 9 discuss sound and artificial intelligence, none of which have actually been implemented yet. Instead they show how an implementation could be used in the future and what to think about.

Section 10 describes how the system for levels and maps works. This is the system that tells the graphics engine what to draw and where. The levels also contain all the information used by the physics engine, such as gravity and atmospheric density.

Finally, Section 11 is a conclusion which summarizes the whole thesis.

2 DirectX and COM

The purpose of this chapter is to give the reader a brief introduction to the basic concepts of the DirectX *Application Programming Interface*³ (API) and Microsoft *Component Object Model* (COM).

The chapter comprises:

- An introduction to DirectX
- An introduction to COM
- How DirectX and COM relate

2.1 The DirectX Kernel

DirectX is an abstraction of many sub components that allows the programmer to utilize any hardware that is DirectX compatible. Rather than creating a module for each hardware manufacturer (as was common a few years ago), a programmer may use DirectX to get working code for virtually any home computer configuration. DirectX is a single component, controlling the communication with all hardware in a faster and more stable way than regular Windows components, such as *Graphics Device Interface*⁴ (GDI) and *Media Control Interface*⁵ (MCI), (see Figure 2.1) which are standard Windows libraries for graphics and sound.

The functionality of DirectX is such that it gives the programmer close to direct control of the hardware. DirectX achieves this by using a multitude of libraries and drivers — written by both Microsoft and the companies who design and construct hardware. Microsoft defined a set of data structures and algorithms to be used by the hardware creators' programmers when they make DirectX compatible drivers for their components. This way, the

³A calling convention by which a program can access services such as the file system or monitor. It is used to abstract underlying logic, thus enabling portability.

⁴The Windows standard COM object for graphics.

⁵The Windows standard COM object for sound.

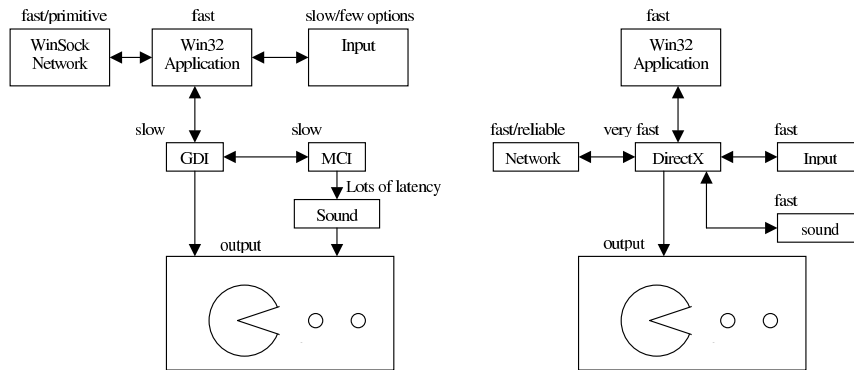


Figure 2.1: “Classical” Programming vs. DirectX

only functions that are being called are DirectX functions, and the application programmer has no need to know what happens at the backend.

With DirectX 8.0, Microsoft integrated DirectDraw and Direct3D into a single component; DirectX Graphics. DirectDraw and Direct3D still exist, but they are no longer updated. The same applies to DirectSound, DirectSound3D and DirectMusic, which are now bundled into DirectX Audio. A great advantage of this system is complete backwards compatibility; any program that works with old versions of DirectX will work with new versions. Thanks to the usage of COM there’s not even a need to recompile programs when a new version of DirectX is released. Another advantage is that when you learn one version of DirectX, you basically know them all. New versions don’t alter old functions, merely add more functionality.

2.2 HAL and HEL

Figure 2.2 shows a couple of things that have not been mentioned thus far: *Hardware Abstraction Layer* (HAL) and *Hardware Emulation Layer* (HEL). Their purpose is to fill any gaps in hardware drivers. If some hardware component does not support a certain function, HAL and HEL will take care of it.

HAL is, as the name suggests, a layer that communicates directly with the hardware. DirectX makes sure that available hardware is used whenever possible, to minimize the load on the *central processing unit*⁶ (CPU). The HAL implementations lie within the drivers of the hardware.

When HAL cannot be used for a specific task, due to hardware restrictions, HEL will emulate hardware capabilities to let the CPU take care of the calculations instead. This results in slower, but working code, and is vital to ensure universal compatibility. Compared to the 80s, programmers have an easy job creating games with DirectX.

2.3 The Components of DirectX

Figure 2.2 shows how the components of DirectX are interconnected, below are semi-detailed descriptions of each component.

2.3.1 DirectDraw

DirectDraw is the main rendering device of bitmap⁷ graphics. DirectDraw also controls the graphics memory, which is the main medium through which all graphics must go before it can be displayed on the monitor. DirectDraw more or less controls the graphics card. DirectX 8.0 and above don't use DirectDraw as a component in its own right, but runs everything through DirectX Graphics.

⁶The part of a computer that does most of the data processing; the CPU and the memory form the central part of a computer, to which the peripherals are attached.

⁷The most basic of image formats, stores each pixel as three byte of data; one for each of the colours Red, Green and Blue.

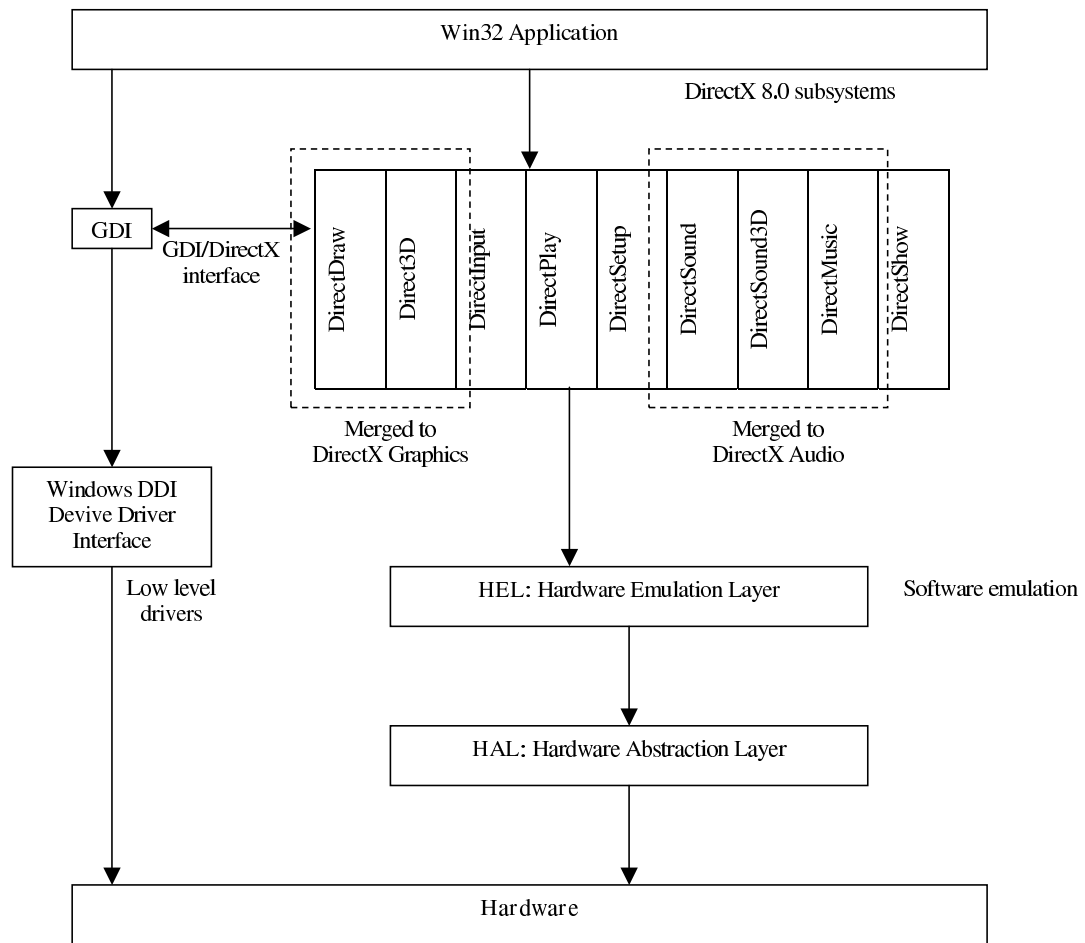


Figure 2.2: DirectX and Windows

2.3.2 DirectSound

This component helps standardising sound playback. Before DirectSound existed, sound programming was black magic where every sound card manufacturer supplied their own drivers and the game programmers had to make a routine for each card. Remember old games where you had to chose sound card, IRQ and DMA? Thanks to DirectX you don't have to do that anymore.

2.3.3 DirectSound3D

The 3D version of DirectSound allows you to place sound sources in a room, the same way you do with regular 3D objects. When you move around in the room the sound will change according to your movements. DirectSound3D supports, among other things, reflection, refraction and the Doppler effect.

2.3.4 DirectMusic

The component handles the music in your games is called DirectMusic. Mainly MIDI⁸, MP3⁹ and CD tracks are used.

2.3.5 DirectInput

DirectInput handles all the input from devices such as the keyboard, mouse or joystick. It supports Force Feedback¹⁰. At the moment there's no support for speech recognition or voice control, but this would be the logical place to put that when it arrives.

⁸A standard for representing musical information in a digital format. MIDI does not use recorded sound for playback, but rather mathematical formulae for how each note should sound for every instrument.

⁹MPEG-1 layer 3. An audio compression standard that can compress CD-tracks about ten times with very little quality loss.

¹⁰A standard for input devices that allows the user to feel a fairly natural response from the game.

2.3.6 DirectPlay

This is the network communications component of DirectX. DirectPlay allows you to create abstract connections with other computers with an IP address through any medium. You don't have to know much, if anything, about network programming to be able to use DirectPlay properly. There's no need to understand the protocol stack, or know which socket to use.

DirectPlay supports *sessions* and *lobbies*. A session is an ongoing network game and a lobby is a server connection that you connect to inbetween games.

2.3.7 Direct3D

The 3D graphics part of DirectX is split into two subcomponents; *Direct3D Retained Mode* (Direct3DRM) and *Direct3D Immediate Mode* (Direct3DIM). RM is a basic high level system that is relatively simple, but slow. IM is a low level system that is alot more complicated, but also alot faster. The science of 3D graphics optimization is well outside the scope of this document.

2.3.8 DirectSetup

DirectSetup is used to simplify the installation of DirectX components. DirectX is very complex, and therefore hard to install manually.

2.3.9 DirectX Graphics

With version 8.0 of DirectX, Microsoft combined DirectDraw and Direct3D to enhance performance and allow 3D effects in a 2D environment. Both DirectDraw and Direct3D still exist and work, but to call them you have to use the interface from before version 8.0.

2.3.10 DirectX Audio

As with DirectX Graphics, Microsoft combined the sound components DirectSound, DirectSound3D and DirectMusic into DirectX Audio. The old components still exist for backwards compatibility.

2.3.11 DirectShow

This is the component that is used for playing video in DirectX. It will automatically locate any hardware acceleration and use it if it exists. This component is very useful since you don't have to care about anything but loading the film and playing it on the monitor.

2.4 COM

Computer programs of today can contain millions of lines of code. Red Hat 7.1, for example, contains over 30 million lines of code [15]. The sheer size of this code requires a structured hierarchy and data abstraction to avoid chaos.

COM, short for Component Object Model, is one solution to this problem. COM is designed to be modular, like pieces of Lego or microchips. They work the same way, no matter what you connect them to. They are all modular and they care only about what input they get, not what sent it to them. In the same way, COM doesn't care what language is used to send it input, as long as it gets input it knows how to deal with. This modularity comes with a few nice advantages; It provides for easy re-implementation of a component — since any COM object with the same interface can replace another — and it allows components to be programmed in different languages, thus giving the developers a chance to utilize the language that is best for any given task.

Yet another step towards modularity comes from the fact that COM objects are compiled into *Dynamically Linked Library* (DLL) files which are loaded during execution rather than during compilation, this means that components can be exchanged without meddling

anything with the main program.

2.4.1 The COM Objects

A COM object is a class that implements a number of interfaces used to communicate with it. The most basic object implements the interface IUnknown which doesn't really do anything. For an object to be useful, however, the programmer has to implement at least one interface of his own and add functionality to it.

Since the objects are completely binary¹¹, and have a uniform calling convention, it doesn't matter what programming language they are coded in, nor does it matter what language they are called from. They will always behave predictably. One of the authors waits for the day when DirectX and COM are ported to Linux so games will be platform independent, whereas the other claims that this could already be the case if only developers used *OpenGL*¹² and *SDL*¹³

2.4.2 GUID

To create a COM object a *Global Unique Identifier* (GUID) is needed. A GUID is a 128 bit long integer that is created by the operating system. The integer is divided into a four byte word, three two byte words and six single byte words [5]. The whole GUID is always unique within the system and is used to distinguish objects. A GUID is assigned to a COM object, during runtime, by the operating system itself. The programmer doesn't have to concern him- or herself with it. As mentioned earlier in this thesis, the authors won't go into detail regarding the COM objects, but this introduction is included because DirectInput uses GUIDs to locate hardware (keyboards, mice, joysticks, hand controls etc).

¹¹As opposed to containing any code.

¹²Open Graphics Library.

¹³Simple DirectMedia Layer. The SDL implementation in Windows uses DirectX for input and sound.

2.5 DirectX and COM

DirectX comprises a large number of COM objects. These objects are located in DLL files that are loaded when a DirectX program is started. Once in the memory the DirectX program can begin to use their interfaces which in turn use their methods to manipulate data.

When Microsoft created DirectX they had to offer more than efficient execution; The game programmers wanted the API to be as easy to use as possible. Since COM isn't a very nice interface to work with, MS encapsulated about 90 % of all COM calls in DirectX functions, thus hiding the fact that it's COM one has to deal with.

Compiling a DirectX program requires a number of library files and associated header files. Each component in DirectX usually has a lib file and a header file, but these files don't contain the actual COM object, but rather shells and references to the DLL files, which do. DirectX objects are almost exclusively called using a function pointer (or function reference). The value of this pointer is set during runtime, not compilation, and is yet another step to modularization and hardware independence.

3 Game Engine Structure

Figure 3.1 shows a simple overview of the modules used within the engine. The first step is initialization which then allows the main event loop to take control and start executing game instructions.

The main event loop — consisting of input handler, AI executioner, physics simulator and graphical engine — continues until the exit command is given by the user, most commonly by pressing the escape key.

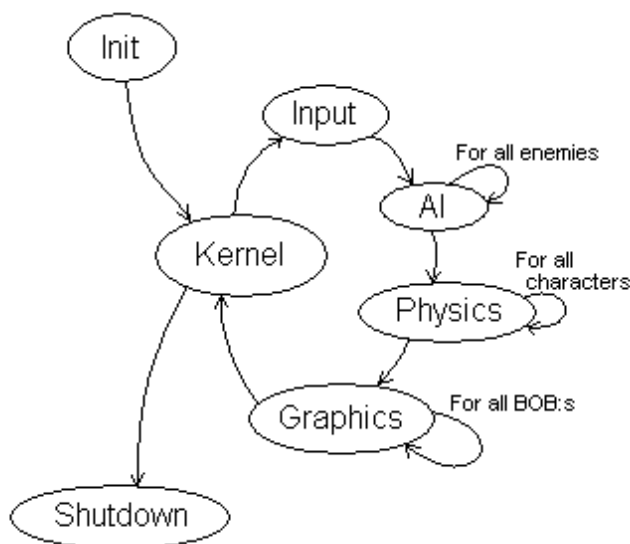


Figure 3.1: Module Overview

The only purpose of the main event loop (also known as the Kernel) is to tell the other objects when it's their turn to execute, and to make sure that each frame¹⁴ within the game takes place at its correct time.

Input is handled by the Player object. DirectInput is called to check if any of the used keys are pressed, and variables are set with corresponding values. For example, if the left

¹⁴A frame is one image on the screen. When a game is played, a multitude of frames are shown every second to ensure smooth movements.

key is pressed, the variable `walk_force` is set to a value appropriate for walking to the left.

When this is done, the physics engine is called into action. The values set by the input are calculated into accelerations which then affect the velocities of the object in question (mainly the player character, but also weapons he might fire or other objects that he can move).

After the player is done, the computer takes care of all of its own characters with artificial intelligence. This is similar to the input stage, but instead of reading from input devices such as a keyboard, the computer executes predefined AI functions for all of its characters. The AI physics is handled the same way as player physics. In fact, they're most likely the same function, inherited from the Character class¹⁵.

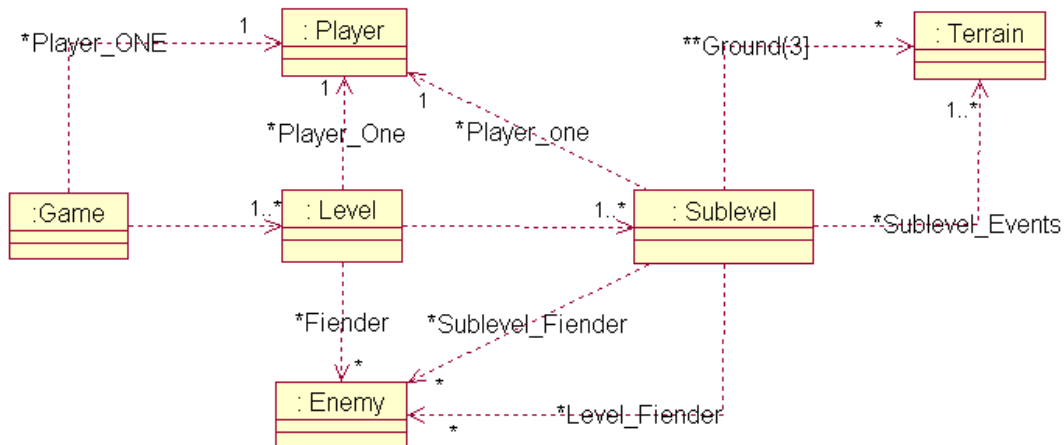


Figure 3.2: Object Diagram of the Game Structure

Note in Figure 3.2 that Game, Level and Sublevel all contain the same player object. Also note that the pointers *Fiender* in Level and *Level.Fiender* in Sublevel both point to the same memory area.

Once the velocities have been set, the physics engine moves all the objects to their new positions and check for collisions. Collisions are treated depending on the objects that

¹⁵Enemy units can have uniquely implemented physics functions if they aren't supposed to adhere to the normal laws of the world.

collide; a player colliding with a wall will simply stop, but characters colliding with bullets will take damage.

After the objects have moved, they will be dispatched to the graphics engine where they are drawn on a surface which then replaces the image that is currently on the screen.

It's now time to check for new input and repeat until the program is terminated.

4 The Kernel

The kernel is the module of the game that keeps track of all other modules. It tells them when to execute and makes sure they're all working in the right order. In the engine, the kernel consists of the object `Game`. `Game` has three member functions; `Init`, `Main` and `Shutdown`. `Init` is used to start DirectX and set state variables within the engine. `Main` is a simple state machine that keeps track of what level the player is currently in. `Shutdown` closes DirectX and deallocates all dynamic memory. The function `Game_Main()` is repeatedly called from within `WinMain()`¹⁶ (see Appendix B). Each call to `Game_Main()` represents a single frame update in the game (sometimes called a 'tick').

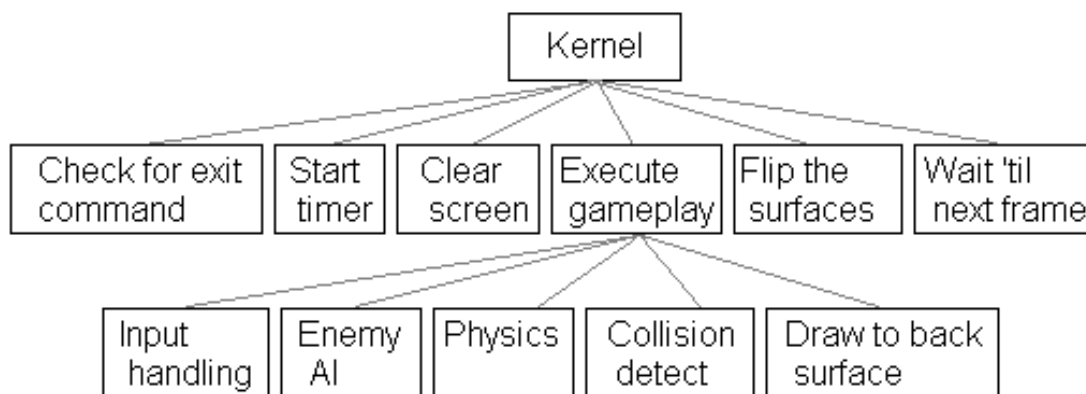


Figure 4.1: Game Kernel structure

`Game_Main()` in turn calls the different modules that make up the game. Figure 4.1 shows a simplified call tree with time increasing from left to right and, to a lesser extent, from top to bottom. Most of the individual blocks will be described in the following sections.

The function is basically a finite state machine that chooses which sub event loop should be called. The object `Game` (see Appendix E.7) has member data to supervise the player and levels. One of the members is of type `Level` (see Appendix E.6), which is a master

¹⁶the main function in a Windows program

level with a number of sublevels (Appendix E.5). These sublevels contain the actual maps that the players can see and interact with.

The Level object has a function called `Playing_Level()`. The function acts like a state machine, deciding what sublevel to call. `Playing_Level()` has to be called once for each frame, just like `Game_Main()`.

The sublevels have a method called `Playing()` which handles input and logic for the game. The sublevel also contains the enemies and events. Some of the enemies are local to the sublevel and will disappear/reappear when the player enters and exits the room, while others belong to the master Level object and can only be killed once. A good example of a similar system is Zelda II [10] where the castles are levels, and all the rooms one can enter are sublevels.

The game logic and physics is implemented in the object Character, which inherits BOB and is inherited by Player and Enemy. The Player object has a pointer to the terrain in sublevel which is used when the player moves. It is not, in fact, the player that moves, but all the surrounding terrain and enemies that are moved around the player which always stays centered on the screen.

Player and Enemy are fairly similar. The main difference is that where Player has the function `Input()` that makes calls to `DirectInput` to do what the player wants it to do, the Enemy class has the member `AI()` to simulate user input. Both classes use the function `Logic()` which handles basic collision detection and movement.

Events, such as doors one can walk through, will be implemented on a case to case basis. These implementations will deal with special animations, sounds and possible movie sequences.

5 Graphics Engine

A graphics engine is the part of a game that simplifies the drawing of things on a screen. Using a graphics API such as DirectX or OpenGL directly is quite arduous with lots of repeated function calls and similar.

The graphics engine used in this program is a rewritten version of André LaMothe's engine described in [1] and [2]. The main difference is that this version is object oriented with classes as the main datatype, rather than structs.

The central object is the *Bitmap Object Blitter* (BOB) (see Appendix D) which loads a bitmap and shows it on the screen.

5.1 Bitmaps

Bitmaps constitute the simplest form of computer graphics. They consist of a matrix of numbers corresponding to colours.

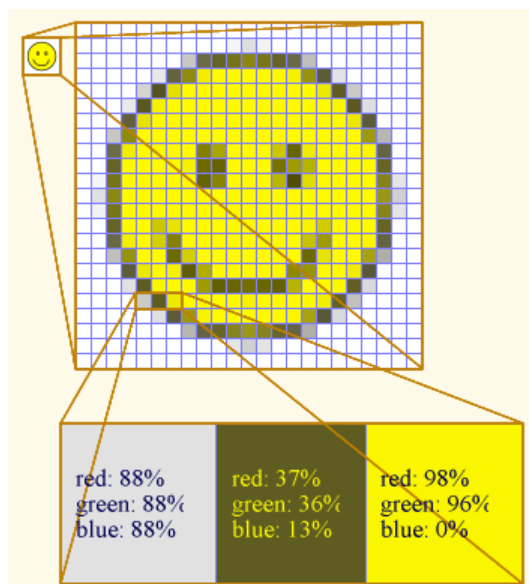


Figure 5.1: Bitmap Example, courtesy of [6]

There are different versions of the bitmap format, among these are:

- 1 bit** Also known as monochrome. Each value in the value matrix points to one of two colours in a small palette. These are usually black and white, but can be changed to other values.
- 8 bit** Like the monochrome format, the values point to palette entries. The difference is that the palette has room for 256 colours rather than just two.
- 16 bit** Two bytes are used to represent the colour of each pixel. Each prime colour uses five bit each, so the format actually only uses 15 bit, not 16. 24 bit graphics can be converted to 16 bit by dividing each of the bytes with 8 and then appending the values to each other in a two byte variable (`col16bit = ((red8bit >> 3) << 10) + ((green8bit >> 3) << 5) + (blue8bit >> 3);`).
- 24 bit** Each pixel is made up of three bytes representing the colours red, green and blue. These prime colours are then combined, with values from 0 to 255, to make just about any colour possible.

Animations with bitmaps are accomplished by loading a number of images to the video memory and then switching between them rapidly. All these images are stored in the same bitmap which is partitioned into squares.

The Bitmap class is used to read data from a file and is then used as an argument for loading graphics into a BOB.

Look at the end of the first page of Appendix D for a class definition of the bitmap.

5.2 Bitmap Object Blitter

As mentioned previously, BOB is the central image component of this engine (See Appendix C and D). It is an object oriented version of the struct created by André LaMothe. BOB is an abstraction on top of DirectDraw and is designed to implement a more user

friendly interface. The class uses DirectX by creating a DirectDraw object and two drawing surfaces which are used to make sure the drawing of a new frame is smooth.

BOB takes care of the coordinates at which a bitmap will be drawn, what image should be shown, how quickly images should change and in what direction and velocity the object is moving. BOB also draws the image on the surface.

When `Draw()` is called, BOB simply tells DirectDraw where to draw, and what to draw by using the `Blit()` method of a DirectDraw surface.

Figure 5.2 shows how BOB is inherited by the other graphical components of the game. Character is the base of all movable objects and is inherited by Player and Enemy which are somewhat more specialised. Player is the object that handles input from a human player, and Enemy is an abstract class that is inherited by all computer controlled characters in the game. Enemy contains a virtual function `AI()`, which replaces the `Input()` function of the Player class. Rullare and Hoppare are the two enemy types used in the demo games created for this report.

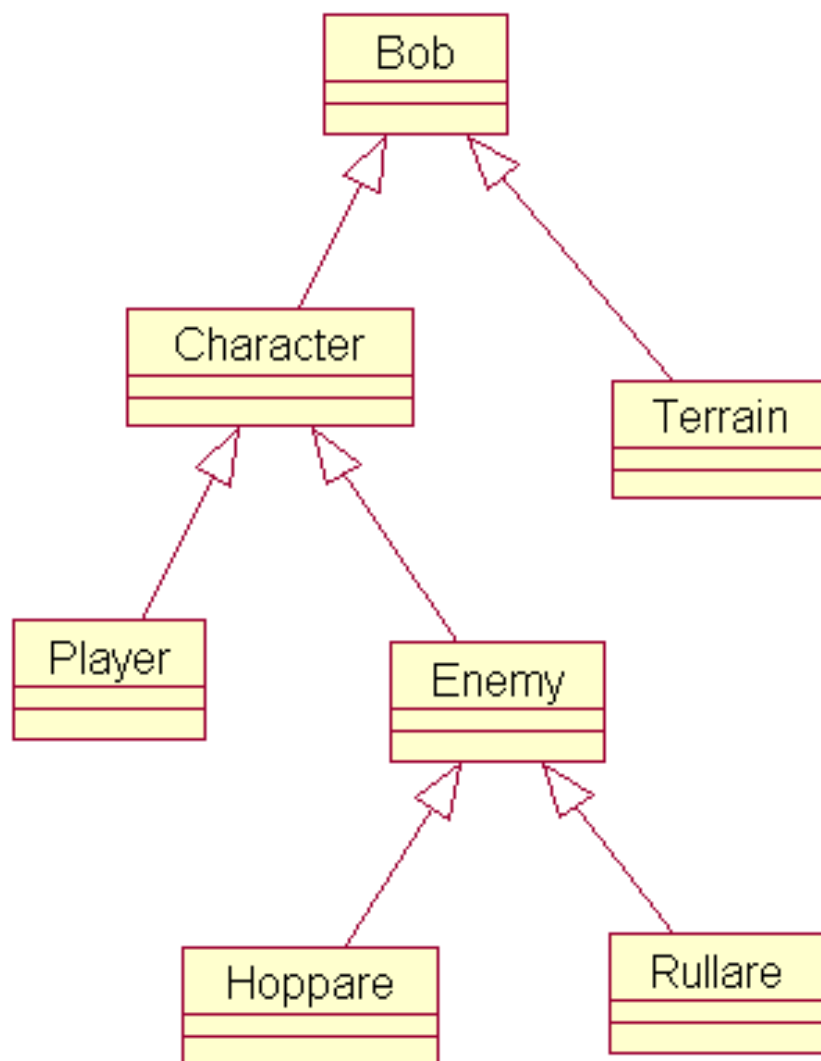


Figure 5.2: BOB Inheritance

6 Physics

For a game to be satisfyingly realistic, it will have to use realistic physics. This means that the movements of characters, monsters, projectiles and similar will have to adhere to the same laws of physics that act in the real world. This chapter will explain what laws will be needed, why they are needed, and how they apply to the objects in a game.

Most formulae in this chapter are taken from [4] and [7]

1. *Law of Inertia* — If no force is acting on a body, it will remain at rest or move in a straight line at a constant velocity.
2. *Law of Acceleration* — The acceleration of a body is proportional and in the same direction as the resultant force that is acting on it.
3. *Law of Action and Reaction* — For any action (force) on a body, there is an equal and opposite reaction (reacting force).

6.1 Newtonian Mechanics

Newtonian, or classical, mechanics is the first part of physics most people will learn in school. It deals with how forces interact with bodies and is based on Newton's laws of motion:

6.1.1 Movements — Acceleration and Inertia

To alter the velocity of an object, in any direction, the application of a force is required. the change in velocity is proportional, and equal in direction, to the force according to the following:

$$m \mathbf{a} = \mathbf{F}$$

Example code:

```
// Temporary velocity variable used to calculate the drag  
float tempvx = xv + walk_force / mass;
```

6.1.2 Drag

When objects move through gases or fluids, the surrounding environment acts as a resistance to slow them down. There are two formulae to describe this resistive force:

$$\mathbf{F}_v = -C_f(0.5 \cdot \rho \cdot \mathbf{v}A)$$

and

$$\mathbf{F}_v = -C_f(0.5 \cdot \rho \cdot \mathbf{v}^2 A)$$

where \mathbf{F}_v is the viscous drag force, C_f is the fluid drag coefficient, \mathbf{v} is the speed of the object, ρ is the density of the air, A is the cross section area of the moving object and the minus sign means that the force works in the opposite direction of the speed. The first formula is valid for objects moving slow enough not to cause any turbulence in the fluid or gas (currently not used in the game engine). The second is for fast moving objects that causes the flow streamlines surrounding it to become disturbed. C_f is usually not equal for these two formulae.

Example code:

```
float dragx = -(0.5f * sublev->rho * tempvx * tempvx * area)  
             * Cd * sign(tempvx);
```

6.1.3 Free-fall

Objects falling towards a large body will experience acceleration due to gravity, the general formula for this is

$$\mathbf{F} = G \cdot \frac{m \cdot M}{\mathbf{r}^2}$$
$$m\mathbf{a} = G \cdot \frac{m \cdot M}{\mathbf{r}^2}$$
$$\mathbf{a} = G \cdot \frac{M}{\mathbf{r}^2}$$

where \mathbf{F} is the force acting on both bodies, G is Newton's Universal Constant (gravitational constant), m is the mass of the smaller object, M is the mass of the large body and \mathbf{r} is the distance between the center of gravity of both bodies. The vector \mathbf{a} is the acceleration resulting from the force.

Example code:

```
// Check to see if object is on the ground
if (On_Ground()&ON_GROUND)
{
    // the ground sets the gravity acceleration to 0
    yv=0;
}
else
{
    // is it touching the roof?
    if (On_Ground()&HITTING_ROOF)
    {
```

```

        // set speed downwards to 1g/s
        yv=sublev->GRAVITY;
    }
    else
    {
        // apply regular gravity
        yv+=sublev->GRAVITY;
    }
}

```

6.2 Physics Within the Game

The formulae described above are used in the game engine to make objects move as naturally as possible. The most basic is the gravitation that pulls all objects towards the ground, this is implemented using actual gravitational constants and masses for all objects and the resulting acceleration is stored in the Level objects.

The next part is drag and user input. Both these instances cause force variables to be set to certain values. The drag calculations are performed with the actual area of the object in question, aswell as proper density and pressure values.

When all forces have been correctly setup the physics engine will calculate the proper acceleration and alter the velocity variables xv and yv (Appendix D) accordingly. Figure 6.1 shows a player character with the velocity vector actually drawn next to it. This was only used in debug mode, the vector does not show in normal gameplay.

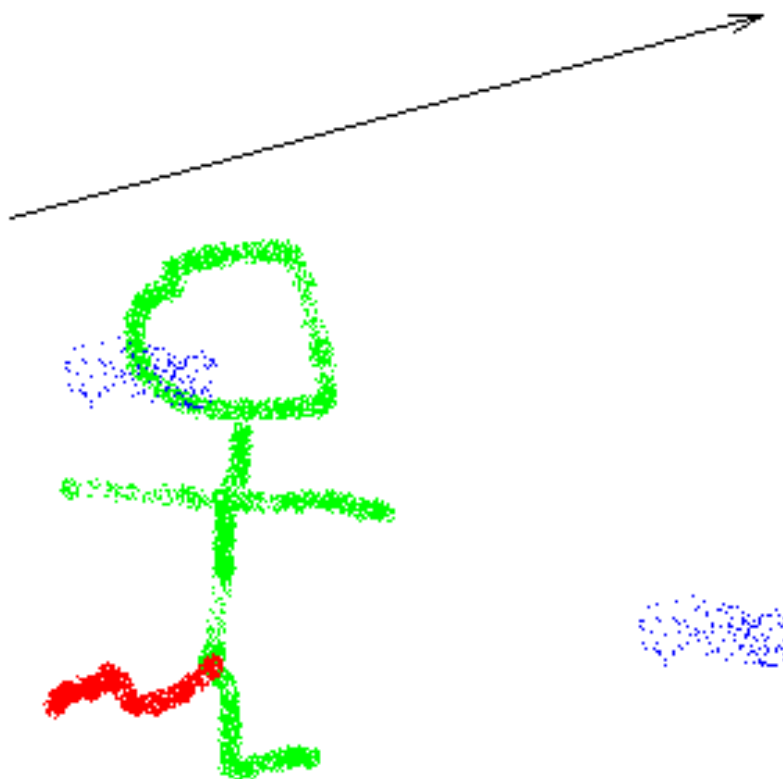


Figure 6.1: Velocity Vector

Example code:

```
// The actual velocity variable  
xv += (int)((walk_force + dragx) / mass);
```

Finally, there's collision detection which prohibits objects from falling through the ground they're being pulled towards. When an object touches a piece of ground, its velocity along the y axis is set to 0. When touching the ceiling, the same velocity vector is set to 0 and gravity kicks in. Collision with walls are handled the same way as with the ground, but it's the velocity along the x axis that is affected.

6.3 Collision Detect

There must be a way to prevent objects from moving through walls, floor and ceilings, as well as making sure that something happens when objects collide with each other.

The BOB object has a function which will check if two BOBs are overlapping. There is, however, another problem.

6.3.1 Running Through Walls

Keeping track of the speed and direction of a moving object in a game can easily be done with a velocity variable for each axis. For each frame, the position of the object will change as many pixels as the velocity variables state in each direction.

Now, imagine a 20 pixel wide object moving at the speed 200 pixels per frame towards a wall (Fig. 6.2 a). The object is 50 pixels away from the wall, which is 50 pixels thick. Let's move one frame forward; without collision detect, or with collision detect only on the position where the object ends up, it would move *through* the wall and stop 80 pixels farther away (Fig. 6.2 b).

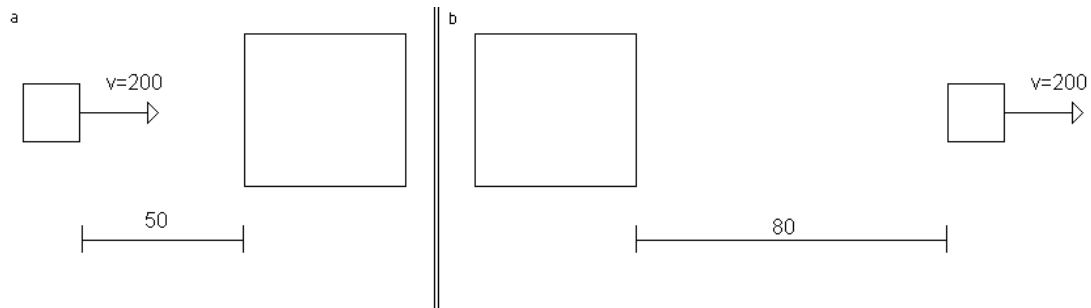


Figure 6.2: Collision Example

This is, of course, not acceptable in a game claiming to have a decent physics engine. Imagine how irritated a player (Fig. 6.3 a) would be if he went straight through a ledge he aimed for (Fig. 6.3 b), rather than stopping at it (Fig. 6.3 c). Or, possibly worse, imagine people running through all the walls from the beginning of a level to the end, cheating their way through a whole game.

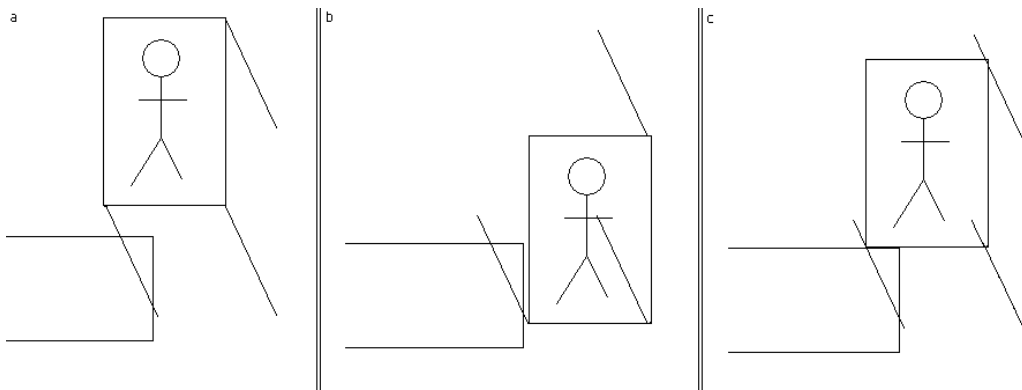


Figure 6.3: Another Collision Example

A simple solution would be to limit the maximum speed, but that wouldn't conform well with the idea of a realistic physics engine.

The solution chosen for this engine is to move the object one pixel at a time, following a line from its start location to the end point, and perform a collision detect on every step. This way a collision is discovered the first time a pixel is overlapping another, and proper

actions can be taken.

Of course, a good line approximation algorithm is essential to this approach.

6.3.2 The Bresenham Algorithm

Drawing exact lines on a computer is impossible, since lines are defined as an infinite number of zero-area points which lie between two end points. The smallest unit on a computer screen is the pixel, and its area is quite a lot more than zero. Approximations on the other hand, are quite easy to draw if you use floating point operations:

```
void draw_line(int x1, int y1, int x2, int y2)
{
    int dx = x2 - x1;
    int dy = y2 - y1;
    float m = dy / dx;
    for (int x = x1; x < x2; x++)
    {
        int y = m * x + y1 + 0.5;
        putpixel(x, y);
    }
}
```

This, however is too slow to be acceptable in a game where speed is of the essence. The solution is Bresenham's midpoint algorithm [13] that will compute the point coordinates correctly, using only integer math.

Figure 6.4 shows how pixel positions (the dots in the corners of the grid) are chosen depending on the true line's (the line between the bottom left corner and the top right

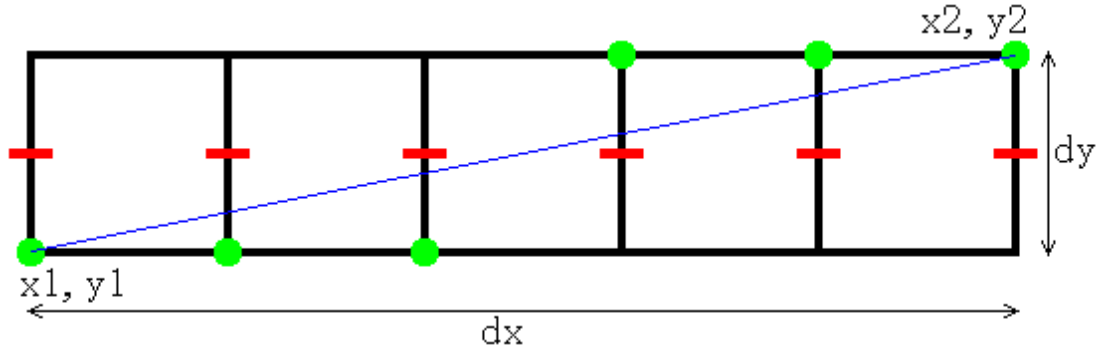


Figure 6.4: Line Approximations

corner of the grid) position relative to a midpoint (short horizontal lines). If the blue line is below the current midpoint, we plot the next pixel to the right. If, however, the blue line is above the midpoint, we should plot above and to the right:

```

If ( BlueLine < Midpoint )
    Plot_Right_Pixel();
Else
    Plot_AboveRight_Pixel();

```

Now to determine if a line is above or below a midpoint at the specific point. To do this we work a bit with the line function $y = kx + m$ where we replace k with dy/dx :

$$y = \frac{dy}{dx} \cdot x + m,$$

$$dx \cdot y = dy \cdot x + dx \cdot m,$$

$$0 = dy \cdot x - dx \cdot y + dx \cdot m.$$

Any point (x, y) above the line will give a negative result, and a point below the line

will give a positive result. We use this fact to define a function F such that

$$F(x, y) = 2 \cdot dy \cdot x - 2 \cdot dx \cdot y + 2 \cdot dx \cdot m,$$

the factor 2 will become evident shortly.

When provided with a midpoint between the two possible next points the function F will return < 0 when the true line is below the midpoint and > 0 when the true line is above.

If we start at point (x_1, y_1) the next point will be either $(x_1 + 1, y_1)$ or $(x_1 + 1, y_1 + 1)$ so the reference midpoint will be $(x_1 + 1, y_1 + 1/2)$. In order to determine which point is the best approximation we evaluate F for the midpoint:

$$F\left(x_1 + 1, y_1 + \frac{1}{2}\right) = 2 \cdot dy \cdot (x_1 + 1) - 2 \cdot dx \cdot \left(y_1 + \frac{1}{2}\right) + 2 \cdot dx \cdot m$$

Since $F(x_1, y_1) = 2 \cdot dy \cdot x_1 - 2 \cdot dx \cdot y_1 + 2 \cdot dx \cdot m = 0$ we conclude that

$$\begin{aligned} F\left(x_1 + 1, y_1 + \frac{1}{2}\right) &= \cancel{2 \cdot dy \cdot x_1} + 2 \cdot dy - \cancel{2 \cdot dx \cdot y_1} - dx + \cancel{2 \cdot dx \cdot m} \\ &= 2 \cdot dy - dx \end{aligned}$$

This is the initial decision variable and will be designated d_0 . Changes in midpoint values can be calculated using $incR = F(Mx + 1, My) - F(Mx, My) = 2 \cdot dy$ when the pixel is to the right and $incUR = F(Mx + 1, My) - F(Mx, My) = 2 \cdot dy - 1 \cdot dx$ when it's to the right and up. This means that all that has to be done to figure out what the next pixel should be is to add $incR$ or $incUR$ to the current d_i value and check if it is positive or negative. See [13] for more details.

7 Input

A player has to be able to interact with the game or the game will have no purpose whatsoever. The interaction between the player and the game is called input. So far, only keyboard input is supported by this game engine, but eventually it will also take care of mouse events and joysticks (including gamepads).

Player input is handled by the Player class. The `Input()` function uses input functions through wrapper macros such as `KEY_DOWN()`:

```
#define KEYDOWN(vk_code) (( GetAsyncKeyState(vk_code) & 0x8000  
    ) ? 1 : 0)
```

7.1 Forces

`Input()` sets physics variables such as `walk_force` in the Player class and the `Logic()` function then calculates movements based on these values.

```
if(KEYDOWN(VK_LEFT)) {  
    // is he not already walking left  
    if(State!=WALKING_LEFT) {  
        // Om stillastående sätt walk_force till  
        walk_beg_left  
        if( On_Ground()&ON_GROUND )  
            walk_force = WALK_BEG_LEFT;  
        else
```

```

                                walk_force = WALK_BEG_LEFT
                                / 4;

                                // set walking left
                                State=WALKING_LEFT;
                                // set face left
                                Direction=FACING_LEFT;
                                // set the walking left animation
                                Set_Animation(1);
                                }
                                }

```

7.2 Impulses

Only once does input alter a movement variable directly; on impulses¹⁷ such as jumping:

```

if (KEY_DOWN(VK.SPACE) )
{
    if ( On_Ground () & ON_GROUND)
        yv=-50; // the initialization speed in the y axis
                from the jump
}

```

¹⁷The change in momentum over a short period of time. The collision between two billiard balls is a good example of an impulse; the energies and momenta of both balls change momentarily.

8 Sound and Music

Even though sound isn't implemented within this engine, it is an important enough subject to merit a brief discussion.

Music and sound effects help giving depth and emotions to a gaming experience. Compare it to the sounds in a movie; the soundtrack can enhance or destroy the whole viewing experience and the sound effects most certainly affect us. A good example is eerie music and sudden sound effects in horror movies.

Old games rarely used digitally sampled sound — since this requires enormous amounts of memory for storage — instead, they used formats such as MIDI which use notation representations to play short sampled sounds, thus creating music.

Later, with the advent of CDs, games began using CD tracks for their music. Eventually, HDD space became cheaper and wav-files were used. At that time the quality of both sound effects and music increased a lot thanks to the capabilities of the sound format.

Nowadays the use of compressed sound formats such as MP3 and OGG are gaining in popularity since they permit a lot of sound to be stored with a small amount of space.

8.1 Sound in the Engine

As mentioned above, there currently is no sound support within the engine. Eventually though, it will be, and should then use the existing DirectX Audio API within a simple wrapper class.

Music will be part of the Sublevel class, possibly to be triggered by events (see Section 10). Ambient sounds will be part of sublevels to be played semi randomly and sound effects will be triggered by events and character actions.

9 The Basics of Artificial Intelligence

Since Artificial Intelligence is a whole science, this thesis won't be able to do more than scratch the surface.

The object `Enemy` has a member function called `AI()` which replaces the `Input()` function of the `Player` object. Since the player orientates himself by looking at the screen, the enemies will have to orientate themselves in a similar fashion. The class `Character`, which `Enemy` inherits, has a reference to the terrain of the level and can use this to “see” what surrounds it.

In his book [3], Steve Rabin includes an article that mentions how to implement intelligent maps with objects that send useful information to computer controlled characters. The game in question was *The Sims*TM, in which items that may provide food for the characters tell the characters about this if they come close. The characters then evaluate their current needs and wants, and if the want for food is high enough they will then approach the object and get something to eat.

A similar feature in this game engine would be to let obstacles and hiding places tell the enemy characters that they may hide there, and if the enemies think that an ambush is appropriate, they will hide. A more advanced AI would make sure that the object is large enough to hide it completely, and try to stand as close to the center as possible to ensure protection.

A simpler implementation of AI would be something that prevents enemies from falling over edges, perhaps by making them stop and change direction, or by making them look for platforms to jump to.

None of the ideas above have been implemented yet, but they are all possible for future versions.

10 The Level System

A graphical game without a playing field of some kind doesn't have much to offer and — unless the game in question is a board game — a game with only a single layout grows old rather quickly. Therefore this game engine comes with a level system and level editor to simplify the creation of new levels.

A level comprises five different parts; foreground, ground, background, enemies and events. foreground, ground and background consist of Terrain objects.

Enemies is an abstract class that is inherited to create moving objects which are controlled by artificial intelligence. The only implementation that has to be done is constructors, destructors, the AI function and, in some cases, the physics and logic functions.

Events are areas on a map which trigger special occurrences when the player enters them. Each event has to be implemented separately to work properly. Events currently use the Terrain class, but that will change in future versions.

Objects are drawn in the following order; background, ground, events, enemies and foreground with background being drawn first and foreground last. The player is drawn concurrently with the enemies. Internally within the layers, objects are drawn in the order they are placed on the map. If a cloud were to be placed after a bush, the cloud would appear in front of the bush.

Objects in background, ground and foreground have an attribute called SpeedFactor. By changing its value, the object's speed relative to the player changes. A value of 0 makes the objects freeze on the screen and follow the player's movements. It is not recommended to change the value of this attribute for objects in the ground layer.

Collision detect is only performed on the ground layer, foreground and background are solely for graphical purposes.

Figure 10.1 shows how the layers are drawn on the screen. The long bar at the bottom is part of the ground layer. The cloud and the left bush are obviously in the background since the player character is partially blocking them. The rightmost bush is in the foreground.

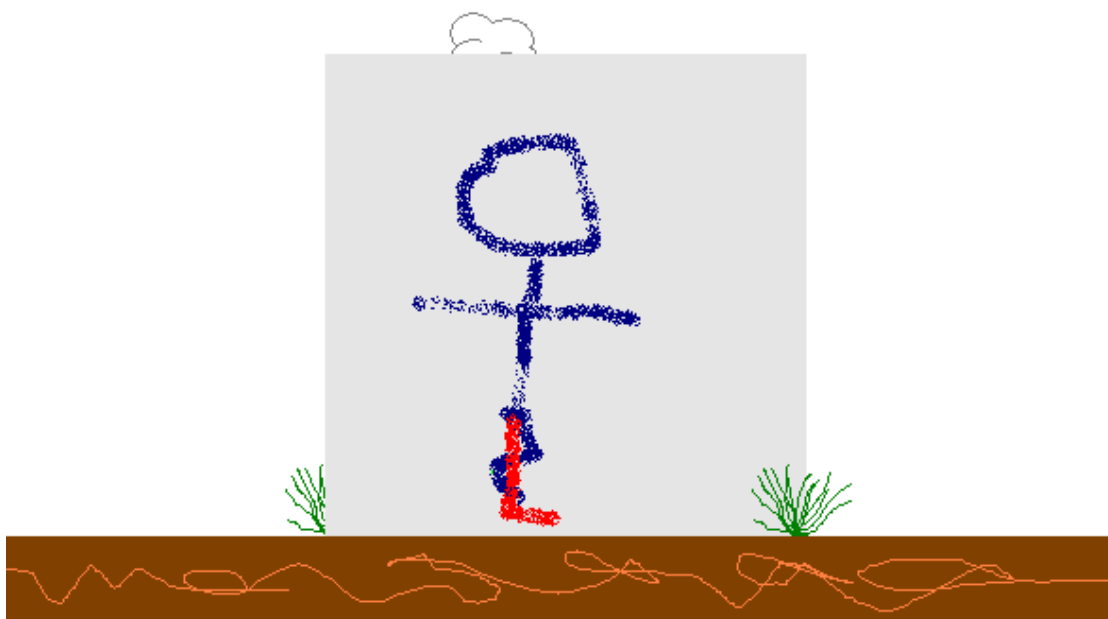


Figure 10.1: The Different Layers

The meta data which describes the objects of the maps are stored in arrays which are allocated and deallocated each time a player enters or exits a sublevel.

DirectX was used for the level editor, simply because this made it easier to save the data to a format that could easily be read by the game engine. Any graphics API could have been used instead, with a little more work.

Some basic instructions for how to use the map editor is shown when the program is executed.

11 Conclusion

The result of the dissertation is a fully working, although pretty basic, game engine with an accompanying level editor used for creating level meta data so one can make new levels. A good system that will be used for the implementation of AI has been created. The system for keeping track of the terrain blocks in the game has been designed and implemented. The terrain blocks inherits the main graphics object, BOB as does the class where the physics is implemented — Character — which is then derived into Player and Enemy.

The level system in the game uses the graphics engine in the sense that all objects on the map use types derived from the base class BOB, but the level objects themselves use no graphics except for debugging purposes.

The Level object contains the character objects, Players and Enemies, which uses the physics and graphics engine. The Input part of the project has been implemented as macros that check if a certain key is pushed or released.

Neither sound, AI nor networking has been implemented, due to lack of time.

11.1 Achievements

- A well documented and working game engine API that allows programmers to create 2D games with a minimum amount of foundation coding, so that they may focus on the gaming experience.
- Mainly succeeded, what is missing in the engine is good graphics, sound effects, music and a good story. Once these elements exist, it could be called a complete platform game.
- A good implementation design that will make it easy to add new features later.
- Done. It is easy to add new terrain types and new enemies to the game. The level editor uses a general system for the maps in the game that is easily modified. The

only coding that needs to be done for each game is for the events, but that can't possibly be avoided.

- A working two dimensional physics engine.
 - Done. No bugs discovered so far.
- A useful level editor that can be used to create the maps of the game.
 - It is useful and general. The level designer needs no knowledge of programming to use it.
- A simple and general way to program the AI of the enemies in the game.
 - AI is never simple when it tries to be good, but the Enemy class has a virtual method called `AI()` that was not implemented in this thesis.
- A virtual class for enemies.
 - Done, derived from character. There are no actual enemies of the class Enemy in the game since all enemies are inherited from the virtual Enemy class.

11.2 Evaluation

The end result is more than satisfactory, The collision detect system in the physics part of the engine uses a much more complex algorithm than was originally planned and the system for maps has pixel precision for placing objects on maps.

There has been no great problems throughout the work on the engine. The goals for the project were set before the implementation and design began so the work has been a more or less straight line from the beginning to the end.

Not all originally planned goals — namely network, sound and AI — were completed, due to lack of time. But this doesn't mean they won't ever be implemented, they have merely been postponed to a later project; a real game using the engine.

11.3 Future Plans

Sound and music haven't played very large parts in the work so far. The very first sound related thing to do in the future, is to add support for sound objects. Music will be added to the Sublevels and possibly be directed by events. Sound effects will belong to the objects responsible for making noise. For example, the sound of gunfire will be stored in the character object that holds the gun.

Ambient sounds can be stored in Sublevel and be played somewhat randomly to enhance the atmosphere of the map. This might include dripping water, the rush of wind in trees and sounds of distant traffic. Transient sounds may be handled by events and played when a player reaches a certain place on the map or performs a task.

The graphical objects Terrain, Characters and its descendants are inherited from BOB. This isn't a very good way to do things since it complicates relocation to another graphics library such as DirectX Graphics or OpenGL. It would be better if the graphical objects included a BOB or similar as a data member. It would also be a good idea to implement characters as a number of smaller BOBs for different parts such as the main body, the weapon, armour etc. This allows the character to change appearance depending on equipment. This obviously assumes that each group of equipment will be fairly uniform so the same animations can be used for at least most of them to save memory and ease implementation.

A nice side effect of such a system would be that it would allow separate collision detect and physics for different parts of the character. This would allow for special vulnerable points where extra damage would be dealt, or part of the character acting as a *mêlée* weapon. The map system wouldn't notice this change since there'd still be a general collision function for the whole object which would be used to detect collisions between the terrain and character.

The map editor has to be altered so objects can be moved "outwards" and "inwards" within a layer. As it is now, they have to be placed on the map in the correct order at

once or they'll be rendered incorrectly. This is a major drawback if one should chose to alter a map.

Player and Enemy should be reimplemented somewhat to make them more similar to the kernel. A common interface used by the kernel would be a good idea, so it can simply store all moving objects in the same list, and call their respective input/AI, physics and graphics functions without discrimination.

References

- [1] André LaMothe, *Tricks of the Windows Game Programming GURUS*, Sams, 2nd Edition, 2002.
- [2] André LaMothe, *Windows Spelprogrammering för DUMMIES*, IDG AB, ISBN 91-7241-006-X, 1999.
- [3] Steve Rabin, *AI Game Programming Wisdom*, Jenifer Niles, 1st Edition, 2002.
- [4] David M. Bourg, *Physics for Game Developers*, O'Reiley & Associates, 1st Edition, 2002.
- [5] Unknown author, *Component object model* — *Wikipedia, the free encyclopedia*, home page, http://en.wikipedia.org/wiki/Component_object_model, 2004-04-28.
- [6] Unknown author, *Raster graphics* — *Wikipedia, the free encyclopedia*, home page, <http://en.wikipedia.org/wiki/Bitmap>, 2004-05-14.
- [7] Carl Nordling and Jonny Österman, *Physics Handbook for Science and Engineering*, Studentlitteratur, Lund, 6th Edition, 1999.
- [8] J. M. Meriam and L. G. Kraige, *Engineering Mechanics Statics SI version*, Wiley, 5th Edition, 2003.
- [9] J. M. Meriam and L. G. Kraige, *Engineering Mechanics Dynamics SI version*, Wiley, 5th Edition, 2003.
- [10] *Zelda II: The Adventure of Link*, Nintendo, 1988.
- [11] Robert A. Adams, *Calculus*, Addison Wesley, 4th Edition, 1998.
- [12] Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley, Special Edition, 2000.
- [13] Hexar, *Drawing Lines — The Bresenham Algorithm*, <http://gamedev.cs.colorado.edu/tutorials/Bresenham.pdf>, 2004-05-10.
- [14] Charles Petzold, *Programming Windows*, Microsoft Press, 5th Edition, 1998.
- [15] David A. Wheeler, *More Than a Gigabuck: Estimating GNU/Linux's Size*, home page, <http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html> 2004-05-30.

A User Manual for the Map Editor

The level editor isn't exactly user friendly, but all functionality is described in a text field on the left hand side of the monitor. Some debuggning information is still printed.

The screen is partitioned into two surfaces; edit area and toolbox. The edit area is the actual map as it will look in the game and the toolbox contains all the objects that can be placed on the level. Figure A.1 shows what the editor looks like before any objects have been placed on the edit area.

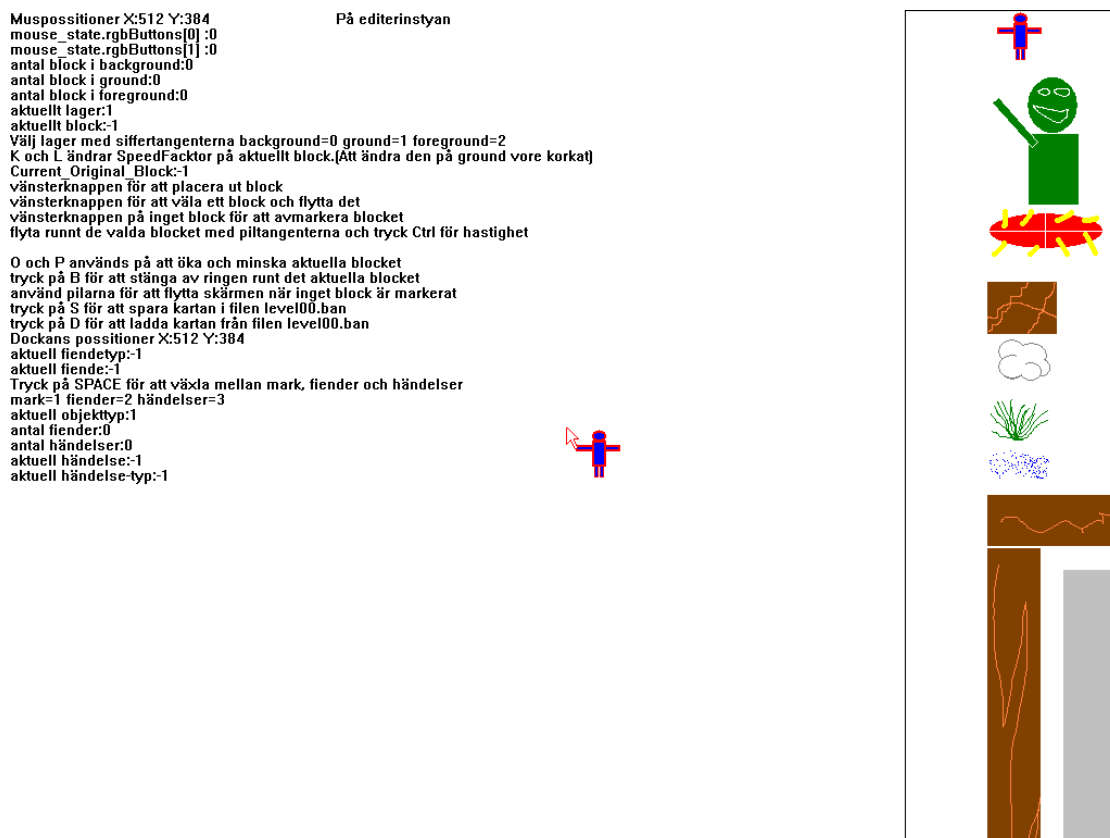


Figure A.1: Empty Level Editor

To begin with, ignore the blue doll. To place an object on the map, left click the object in question in the toolbox and then left click on the edit area where you want it placed.

To move the object to another place, use the arrow keys. O and P are used iterate the list of placed objects.

The space key is used to change between terrain, enemies and events. To specify the terrain layer, use the keys 0, 1 and 2.

To make objects move faster or slower than the player, K and L are used for changing the speed factor. The default value is 1, which means that object is stationary on the map, this is the best value to use for most objects. A value less than 1 makes the object move slower than the player (appearing to be farther away from the “camera”), and a higher value makes it move faster (therefore close to the “camera”). Even the objects in the ground layer can have their speed factor altered, but this isn’t advisable since it would look rather strange.

The view is changed by using the arrow keys when no object is marked. The speed factor isn’t used when moving around in the editor, only within the game itself.

Press S to save the map. It will be called level00.ban and placed in the same folder as the map editor. To open the map level00.ban in the editor, press D.

B winconsole.cpp

```
#define WIN32_LEAN_AND_MEAN
#define INITGUID

#include <windows.h>    // include important windows stuff
#include <windowsx.h>
#include <mmsystem.h>
#include <objbase.h>
#include <iostream.h> // include important C/C++ stuff
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

#include "ddraw.h"      // directX includes
#include "dinput.h"     // directX includes
#include "dsound.h"     // directX includes

#include "game.h"       // My Game Engine

// DEFINES ////////////////////////////////////////

// defines for windows
#define WINDOW_CLASS_NAME "WINXCLASS" // class name

#define WINDOW_WIDTH 64 // size of window
#define WINDOW_HEIGHT 48

// GLOBALS ////////////////////////////////////////

HWND main_window_handle=NULL; // pointer to the window handle
HINSTANCE main_instance=NULL; // pointer to the instance

//////////////////////////////////////
// STANDARD WINDOW FUNCTIONS ////////////////////////////////////////
//////////////////////////////////////

LRESULT CALLBACK WindowProc(HWND hwnd,
                             UINT msg,
                             WPARAM wparam,
                             LPARAM lparam)
{
    // this is the main message handler of the system

    PAINTSTRUCT ps; // used in WM_PAINT
    HDC hdc; // handle to a device context

    // what is the message
    switch(msg)
    {
        case WM_CREATE:
        {
            // do initialization stuff here
        }
    }
}
```

```

        return(0);
    } break;

    case WMPAINT:
    {
        // start painting
        hdc=BeginPaint(hwnd,&ps);

        // end painting
        EndPaint(hwnd,&ps);
        return(0);
    } break;

    case WMDESTROY:
    {
        // kill the application
        PostQuitMessage(0);
        return(0);
    } break;

    default:break;
} // end switch

// process any messages that we didn't take care of
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc

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

int WINAPI WinMain(    HINSTANCE hinstance,
                      HINSTANCE hprevinstance,
                      LPSTR lpcmdline,
                      int ncmdshow)
{
    // this is the winmain function

    WNDCLASS winclass;    // this will hold the class we create
    HWND      hwnd;       // generic window handle
    MSG        msg;        // generic message
    HDC        hdc;        // generic dc
    PAINTSTRUCT ps;        // generic paintstruct

    // first fill in the window class structure
    winclass.style = CS_DBLCLKS|CS_OWNDC|CS_HREDRAW|
                    CS_VREDRAW;

    winclass.lpfnWndProc = WindowProc;
    winclass.cbClsExtra = 0;
    winclass.cbWndExtra = 0;
    winclass.hInstance = hinstance;
    winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    winclass.hbrBackground = (HBRUSH_*) GetStockObject(BLACK_BRUSH);
    winclass.lpszMenuName = NULL;
    winclass.lpszClassName = WINDOW_CLASS_NAME;

    // register the window class
    if(!RegisterClass(&winclass))
        return(0);

    // create the window, note the use of WS_POPUP
    if(!(hwnd=CreateWindow(WINDOW_CLASS_NAME, // class

```

```

        TITLE,                                // title of the window
        WS.POPUP|WS.VISIBLE,
        0,0,                                // x,y
        WINDOW.WIDTH,                        // width
        WINDOW.HEIGHT,                      // height
        NULL,                                // handle to parent
        NULL,                                // handle to menu
        hinstance,                            // instance
        NULL)))                             // creation parms
    return(0);

// save the window handle and instance in a global pointer
main_window_handle = hwnd;
main_instance      = hinstance;

Game Spel;
// perform all game console specific initialization
Spel.Game_Init();

// enter main event loop
while(1)
{
    if(PeekMessage(&msg,NULL,0,0,PMREMOVE))
    {
        // test if this is a quit
        if(msg.message==WM.QUIT)
            break;

        // translate any accelerator keys
        TranslateMessage(&msg);

        // send the message to the window proc
        DispatchMessage(&msg);
    } // end if

    // main game processing goes here
    Spel.Game_Main();

} // end while

// shutdown game and release all resources
Spel.Game_Shutdown();

// return to Windows like this
return(msg.wParam);

} // end WinMain

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

```

C motor.h

```
/*=====
*
* Copyright (C) 2004 Martin Persson. Directly stolen from André LaMothe.
*
* File:      motor.h
* Content:   Game Engine include file
*
*****/

#ifndef __motor_h_
#define __motor_h_

// DEFINES ////////////////////////////////////////

// default screen size
#define SCREEN_WIDTH 1600 // size of screen
#define SCREEN_HEIGHT 1200
#define SCREEN_BPP 8 // bits per pixel

// basic unsigned types
typedef unsigned short USHORT;
typedef unsigned short WORD;
typedef unsigned char UCHAR;
typedef unsigned char BYTE;

#define MAX_SOUNDS 64 // max number of sounds in system at once

#define SOUND_NULL 0
#define SOUND_LOADED 1
#define SOUND_PLAYING 2
#define SOUND_STOPPED 3

// voc file defines
#define NVB_SIZE 6 // size of new voice block in bytes

// screen transition commands
#define SCREEN_DARKNESS 0 // fade to black
#define SCREEN_WHITENESS 1 // fade to white
#define SCREEN_SWIPE_X 2 // do a horizontal swipe
#define SCREEN_SWIPE_Y 3 // do a vertical swipe
#define SCREEN_DISOLVE 4 // a pixel dissolve
#define SCREEN_SCRUNCH 5 // a square compression
#define SCREEN_BLUENESS 6 // fade to blue
#define SCREEN_REDNESS 7 // fade to red
#define SCREEN_GREENNESS 8 // fade to green

// MACROS ////////////////////////////////////////

// these read the keyboard asynchronously
#define KEY_DOWN(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
#define KEY_UP(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 0 : 1)

// initializes a direct draw struct
#define DD_INIT_STRUCT(ddstruct) {memset(&ddstruct,0,sizeof(ddstruct));\
ddstruct.dwSize=sizeof(ddstruct); }

// sets the volume to 0-100 whitout crapy logaritms
#define DSVOLUME_TO_DB(volume) ((DWORD)(-30*(100 - volume)))

// TYPES ////////////////////////////////////////
```

```

// this holds a single sound
typedef struct pcm_sound_typ
{
    LPDIRECTSOUNDBUFFER dsbuffer;    // the ds buffer containing the sound
    int state;                       // state of the sound
    int rate;                        // playback rate
    int size;                        // size of sound
    int id;                          // id number of the sound
} pcm_sound, *pcm_sound_ptr;

// PROTOTYPES //////////////////////////////////////

// DirectDraw functions
int DD_Init(int width, int height, int bpp);
int DD_Shutdown(void);
LPDIRECTDRAWCLIPPER DD_Attach_Clipper(LPDIRECTDRAWSURFACE lpdds,
    int num_rects, LPPRECT_CLIP_LIST);
LPDIRECTDRAWSURFACE DD_Create_Surface(int width, int height, int mem_flags);
int DD_Flip(void);
int DD_Wait_For_Vsync(void);
int DD_Fill_Surface(LPDIRECTDRAWSURFACE lpdds, int color);
UCHAR *DD_Lock_Surface(LPDIRECTDRAWSURFACE lpdds, int *lpitch);
int DD_Unlock_Surface(LPDIRECTDRAWSURFACE lpdds, UCHAR *surface_buffer);
UCHAR *DD_Lock_Primary_Surface(void);
int DD_Unlock_Primary_Surface(void);
UCHAR *DD_Lock_Back_Surface(void);
int DD_Unlock_Back_Surface(void);

// general utility functions
DWORD Get_Clock(void);
DWORD Start_Clock(void);
DWORD Wait_Clock(DWORD count);
int Collision_Test(int x1, int y1, int w1, int h1,
    int x2, int y2, int w2, int h2);
int Color_Scan(int x1, int y1, int x2, int y2,
    UCHAR scan_start, UCHAR scan_end,
    UCHAR *scan_buffer, int scan_lpitch);

// graphics functions
int Draw_Clip_Line(int x0, int y0, int x1, int y1, UCHAR color,
    UCHAR *dest_buffer, int lpitch);
int Clip_Line(int &x1, int &y1, int &x2, int &y2);
int Draw_Line(int x0, int y0, int x1, int y1, UCHAR color, UCHAR *vb_start, int lpitch);
int Draw_Pixel(int x, int y, int color, UCHAR *video_buffer, int lpitch);
int Draw_Rectangle(int x1, int y1, int x2, int y2, int color, LPDIRECTDRAWSURFACE lpdds);
int Screen_Transition(void);
void HLine(int x1, int x2, int y, int color, UCHAR *vbuffer, int lpitch);
void VLine(int y1, int y2, int x, int color, UCHAR *vbuffer, int lpitch);
void Screen_Transitions(int effect, UCHAR *vbuffer, int lpitch);

// palette functions
int Set_Palette_Entry(int color_index, LPPALETTEENTRY color);
int Get_Palette_Entry(int color_index, LPPALETTEENTRY color);
int Load_Palette_From_File(char *filename, LPPALETTEENTRY palette);
int Save_Palette_To_File(char *filename, LPPALETTEENTRY palette);
int Save_Palette(LPPALETTEENTRY sav_palette);
int Set_Palette(LPPALETTEENTRY set_palette);
int Rotate_Colors(int start_index, int colors);
int Blink_Colors(void);

// gdi functions

```

```

int Draw_Text_GDI(char *text , int x,int y,COLORREF color ,
    LPDIRECTDRAW_SURFACE lpdds);
int Draw_Text_GDI(char *text , int x,int y,int color , LPDIRECTDRAW_SURFACE lpdds);

// error functions
int Open_Error_File(char *filename);
int Close_Error_File(void);
int Write_Error(char *string , ...);

// sound
int Load_VOC(char *filename);
int Load_WAV(char *filename ,
    int control_flags=DSBCAPS_CTRLPAN|DSBCAPS_CTRLVOLUME|
    DSBCAPS_CTRLFREQUENCY);
int Replicate_Sound(int source_id);
int Play_Sound(int id , int flags=0,int volume=0, int rate=0, int pan=0);
int Stop_Sound(int id);
int Stop_All_Sounds(void);
int DSound_Init(void);
int DSound_Shutdown(void);
int Delete_Sound(int id);
int Delete_All_Sounds(void);
int Status_Sound(int id);
int Set_Sound_Volume(int id,int vol);
int Set_Sound_Freq(int id,int freq);
int Set_Sound_Pan(int id,int pan);

// input
int DInput_Init(void);
void DInput_Shutdown(void);
int DI_Init_Joystick(int min_x=-256, int max_x=256, int min_y=-256,
    int max_y=256);
int DI_Init_Mouse(void);
int DI_Init_Keyboard(void);
int DI_Read_Joystick(void);
int DI_Read_Mouse(void);
int DI_Read_Keyboard(void);
void DI_Release_Joystick(void);
void DI_Release_Mouse(void);
void DI_Release_Keyboard(void);

// EXTERNALS ////////////////////////////////////////

extern FILE *fp_error; // general error file
extern LPDIRECTDRAW lpdd; // dd object
extern LPDIRECTDRAW_SURFACE lpddsprimary; // dd primary surface
extern LPDIRECTDRAW_SURFACE lpddsback; // dd back surface
extern LPDIRECTDRAW_PALETTE lpddpal; // a pointer to the
// created dd palette
extern LPDIRECTDRAW_CLIPPER lpddclipper; // dd clipper
extern PALETTEENTRY palette[256]; // color palette
extern PALETTEENTRY save_palette[256]; // used to save palettes
extern DDSURFACEDESC ddsd; // a direct draw surface
// description struct
extern DDBLTFX ddbltfx; // used to fill
extern DDSCAPS ddscaps; // a direct draw surface
// capabilities struct
extern HRESULT ddrval; // result back from dd
// calls
extern UCHAR *primary_buffer; // primary video buffer
extern UCHAR *back_buffer; // secondary back buffer

```

```

extern int          primary_lpitch;          // memory line pitch
extern int          back_lpitch;            // memory line pitch
extern DWORD        start_clock_count;      // used for timing

// these defined the general clipping rectangle
extern int min_clip_x,                      // clipping rectangle
          max_clip_x,
          min_clip_y,
          max_clip_y;

// these are overwritten globally by DD_Init()
extern int screen_width,                    // width of screen
          screen_height,                   // height of screen
          screen_bpp;                      // bits per pixel

extern LPDIRECTSOUND lpds;                 // directsound interface pointer
extern DSBUFFERDESC dsbd;                  // directsound description
extern DSCAPS dscaps;                      // directsound caps
extern HRESULT dsresult;                   // general directsound result
extern DSBCAPS dsbcaps;                   // directsound buffer caps

extern LPDIRECTSOUNDBUFFER lpdsbprimary;    // the primary mixing buffer
// the array of secondary sound buffers
extern pcm_sound sound_fx[MAX_SOUNDS];

extern WAVEFORMATEX pcmwf;                 // generic waveformat structure

// directinput globals
extern LPDIRECTINPUT8 lpdi;                // dinput object
extern LPDIRECTINPUTDEVICE8 lpdikey;       // dinput keyboard
extern LPDIRECTINPUTDEVICE8 lpdimouse;     // dinput mouse
extern LPDIRECTINPUTDEVICE8 lpdijoy;       // dinput joystick
extern LPDIRECTINPUTDEVICE2 lpdijoy2;      // dinput joystick
extern GUID joystickGUID;                  // guid for main joystick
extern char joyname[80];                   // name of joystick

// these contain the target records for all di input packets
extern UCHAR keyboard_state[256];          // contains keyboard state table
extern DIMOUSESTATE mouse_state;           // contains state of mouse
extern DIJOYSTATE joy_state;               // contains state of joystick
extern int joystick_found;                 // tracks if stick is plugged in

#endif // __motor_h_

```

D bob.h

```
/*=====
*
* Copyright (C) 2004 Martin Persson. Directly stolen from Andre LaMothe.
*
* File:      bob.h
* Content:   Bitmap Objekt Blitter include file
*
*****/

// watch for multiple inclusions
#ifndef __bob_h_
#define __bob_h_

// DEFINES ////////////////////////////////////////

// defines for Bobs
#define BOB.STATE_DEAD      0 // this is a dead bob
#define BOB.STATE_ALIVE    1 // this is a live bob
#define BOB.STATE_DYING    2 // this bob is dying
#define BOB.STATE_ANIM_DONE 1 // done animation state
#define MAX_BOB_FRAMES     64 // maximum number of bob frames
#define MAX_BOB_ANIMATIONS 16 // maximum number of animation
                               // sequesces
#define BOB_ATTR_SINGLE_FRAME 1 // bob has single frame
#define BOB_ATTR_MULTIFRAME  2 // bob has multiple frames
#define BOB_ATTR_MULTIANIM    4 // bob has multiple animations
#define BOB_ATTR_ANIM_ONCE_SHOT 8 // bob will perform the animation
                               // once
#define BOB_ATTR_VISIBLE     16 // bob is visible
#define BOB_ATTR_BOUNCE      32 // bob bounces off edges
#define BOB_ATTR_WRAPAROUND  64 // bob wraps around edges
#define BOB_ATTR_LOADED      128 // the bob has been loaded
#define BOB_ATTR_CLONE       256 // the bob is a clone

// bitmap defines
#define BITMAP_ID            0x4D42 // universal id for a bitmap
#define BITMAP_EXTRACT_MODE_CELL 0
#define BITMAP_EXTRACT_MODE_ABS 1

// this builds a 16 bit color value
#define _RGB16BIT(r,g,b)      ((b%32) + ((g%32) << 5) + ((r%32) << 10))

// bit manipulation macros
#define SET_BIT(word, bit_flag) ((word)=((word) | (bit_flag)))
#define RESET_BIT(word, bit_flag) ((word)=((word) & (~bit_flag)))

// class definitions ////////////////////////////////////////

// Bitmap
typedef class Bitmap
{
public:
    // this contains the bitmapfile header
    BITMAPFILEHEADER bitmapfileheader;
    // this is all the info including the palette
    BITMAPINFOHEADER bitmapinfoheader;
    // we will store the palette here
```



```

    PALETTEENTRY    palette[256];
    // this is a pointer to the data
    UCHAR           *buffer;

    Bitmap(void);
    int Flip_Bitmap(UCHAR *image, int bytes_per_line, int height);
    int Load_File(char *filename);
    int Unload_File(void);

} Bitmap_File, *Bitmap_File_Ptr;

// Bob
typedef class Bob    // the Bitmap Object Blitter class Bob
{
protected:
    // the bitmap images DD surfaces
    LPDIRECTDRAW_SURFACE images[MAX_BOB_FRAMES];
    // the surface to draw on
    LPDIRECTDRAW_SURFACE *Destination_Surface;

public:
    LPDIRECTDRAW_SURFACE* Get_Destination_Surface(void)
        {return(Destination_Surface);}
    LPDIRECTDRAW_SURFACE* Get_images(void){return(images);}

protected:
    int state;           // the state of the object (general)
    int anim_state;      // an animation state variable, up to you
    int attr;            // attributes pertaining to the object (general)
    int x,y;             // position bitmap will be displayed at
    int xv,yv;           // velocity of object
    int width, height;   // the width and height of the bob
    int width_fill;      // internal, used to force 8*x wide surfaces
    int counter_1;       // general counters
    int counter_2;
    int max_count_1;     // general threshold values;
    int max_count_2;
    int varsI[16];       // stack of 16 integers
    float varsF[16];     // stack of 16 floats
    int curr_frame;      // current animation frame
    int num_frames;      // total number of animation frames
    int curr_animation;  // index of current animation
    int anim_counter;    // used to time animation transitions
    int anim_index;      // animation element index
    int anim_count_max;  // number of cycles before animation
    int *animations[MAX_BOB_ANIMATIONS]; // animation sequences

public:
    // Bob methods
    Bob(int x, int y, int width, int height, int num_frames, int attr,
        int mem_flags, LPDIRECTDRAW_SURFACE *destination_surface);
    virtual ~Bob(void);
    Bob(void);
    int Draw(void);
    int Draw_Scaled(int swidth, int sheight);
    int Load_Frame(Bitmap_File_Ptr bitmap, int frame, int cx, int cy,
        int mode);
    int Animate(void);
    int Scroll(void);
    int Move(void);
    int Load_Animation(int Anim_index, int Num_frames, int *sequence);

```

```

    int Set_Pos(int X, int Y);
    int Set_Anim_Speed(int speed);
    int Set_Animation(int Anim_index);
    int Set_Vel(int Xv, int Yv);
    int Hide(void);
    int Show(void);
    int Collision(Bob *bob2);
    Bob(Bob *bob_src);
} *Bob_ptr;

#endif //__bob_h_

```

E game.h

```
/* *****
 *
 * Copyright (C) 2004 Martin Persson. All rights reserved.
 *
 * File:      game.h
 * Content:   Main Game include file
 *
 * *****/

// watch for multiple inclusions
#ifndef __game_h_
#define __game_h_

// INCLUDES ////////////////////////////////////////
#include "bob.h"

// DEFINES ////////////////////////////////////////

#define TITLE "Martins_Spelmotor" // Title of the window
// terrain layertype defines to keep track of what you are doing
#define BACKGROUND 0
#define GROUND 1
#define FOREGROUND 2

// only used for the ground layer of terran to indicate that pixelscan should
// be used insted of the ordinary Bob::Collision
#define TERRAIN_TYPE_NON_SQUARE 1

#define NUM_ORIGINAL_GROUND 6
#define NUM_ORIGINAL_ENEMIES 2
#define NUM_EVENT_TYPES 1
#define HOPPARE 0
#define RULLARE 1

// character defines
#define STANDING_STILL 0
#define WALKING_RIGHT 1
#define WALKING_LEFT 2

#define FACING_LEFT 1
#define FACING_RIGHT 2

#define IN_FREE_AIR 0
#define ON_GROUND 1
#define HITTING_ROOF 2
#define ON_LEFT_WALL 4
#define ON_RIGHT_WALL 8
#define HITTING_UP_LEFT_CORNER 16
#define HITTING_UP_RIGHT_CORNER 32
#define HITTING_DOWN_LEFT_CORNER 64
#define HITTING_DOWN_RIGHT_CORNER 128

// devierd eneny defines
#define HOPPARE_ANIMATION 0
#define ROLL_RIGHT 0
#define ROLL_LEFT 1

//////////////////////////////////////
// OBJECT DEFINES ////////////////////////////////////////
```

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

E.1 Terrain

```
// TERRAIN CLASS //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
typedef class Terrain:public Bob
{
protected:
    int Terrain_Attributes;
public:
    // used to identify what type of original terrain it was cloned from
    // so the filename can be saved in the *.ban file
    int Type_Index;

    double t_x,t_y;
    double t_xv,t_yv;
    // used to make the ForeGround and BackGround move faster or slower
    // than the player
    double SpeedFactor;

    void Set_Type(int type_arg);
    void Set_SpeedFactor(double SpeedFactor_arg);
    void Set_Pos(int x_arg,int y_arg);
    int Move(void);
    void Move(int X,int Y);

    Terrain(int X, int Y,int Width, int Height,int Num_frames,int Attr,
            int Mem_flags,LPDIRECTDRAWSURFACE *destination_surface);
    ~Terrain(void);
    Terrain(Terrain *terrain_src);
} *Terrain_ptr;
```

E.2 Character

```
// CHARACTER CLASS //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// ska senare ärvas ner till en underklass för spelare och en för varje fiende
typedef class Character:public Bob
{
// this is where the physics engine is implemented
protected:
    // attrubutes
    int Sty,Fys,Smi,Int,Psy,Kar,Sto;
    // skills
    //jag lägger till mer sen!
    // used to know what direction the player is faceing
    int Direction;
    // used to know what the player is doing
    int State;
    // pointer to the ground in the sublevel
    Terrain_ptr **Ground;
    // the size of the map in the sublevel
    int *Terrain_Size;
    // the reference
    Terrain_ptr Referencepoint;
    void Move_Level(int x,int y); // only used by the player
    inline bool Ground_Collision(void);

    int Antal_Handelser_i_Sublevel;
    Terrain_ptr *Sublevel_Events;

    // These values are used for the physics
```

```

    int mass;
    float area;
    float Cd;
    float walk_force;

public:
    Sublevel_ptr sublev;

    void Move(int dx,int dy);

    Character(int X, int Y,int Width, int Height,int Num_frames,int Attr,
              int Mem_flags,LPDIRECTDRAWSURFACE *destination_surface);
    virtual ~Character(void);
    Character():Bob(){} // used for cloneing of enemies
    int Collision(Terrain_ptr Block);
    inline bool Collision_v_1_5(Terrain_ptr Block);
    // returnvalue depends on where the character is
    int On_Ground(void);
    void Set_Ground(Terrain_ptr *Global_Ground[],int size[]);
    void Set_Referencepoint(Terrain_ptr Reference);

    void Set_Sublevel_Events(Terrain_ptr *handelse_arg,int antal_handelser);

} *Character_ptr;

```

E.3 Enemy

```

// ENEMY CLASS ////////////////////////////////////////
typedef class Enemy:public Character
{
    // this is the enemy class
protected:
public:
    Enemy(int X, int Y,int Width, int Height,int Num_frames,int Attr,
          int Mem_flags,LPDIRECTDRAWSURFACE *destination_surface);
    Enemy():Character(){} // used for cloneing of enemies
    virtual ~Enemy(void);
    virtual void Init(void);
    virtual void Enemy_Move(void);
    virtual void AI(void); // motsvarar input i player
    virtual void Logic(void);
} *Enemy_ptr;

typedef class Rullare:public Enemy
{
protected:
    inline bool Reached_Right_Edge(void);
    inline bool Reached_Left_Edge(void);
public:
    Rullare(LPDIRECTDRAWSURFACE *destination_surface);
    Rullare(Rullare *Rullare_src);
    Rullare(Enemy *Enemy_src);
    Rullare():Enemy(){}
    virtual ~Rullare(void);
    virtual void AI(void);
} *Rullare_ptr;

typedef class Hoppare:public Enemy
{
protected:
public:

```

```

    Hoppare(LPDIRECTDRAW_SURFACE *destination_surface);
    Hoppare(Hoppare *Hoppare_src);
    Hoppare(Enemy *Enemy_src);
    Hoppare():Enemy(){}
    virtual ~Hoppare(void);
    virtual void AI(void);
} *Hoppare_ptr;

```

E.4 Player

```

// PLAYER CLASS //////////////////////////////////////
typedef class Player:public Character
{
// this is the Player class
protected:
    void Init(void);
    // used fore the movement of enemies when the player is moving
    Enemy_ptr *Sublevel_Fiender;
    int Antal_Fiender_i_Sublevel;
public:
    Player(int X, int Y,int Width, int Height,int Num_frames,int Attr,
           int Mem_flags,LPDIRECTDRAW_SURFACE *destination_surface);
    virtual ~Player(void);
    void Input(void);
    void Logic(void);
    void Player_Move(void);
    void Set_Enemies(Enemy_ptr *Fiende_arg,int antal_fiender);
    void Move_Level(int x,int y);
} *Player_ptr;

```

E.5 Sublevel

```

// SUBLEVEL CLASS //////////////////////////////////////
typedef class Sublevel
{
protected:
    // Terrain in dynamic arrays
    Terrain_ptr *Ground[3];

    // the original terrain that will be used to clone all other terrain
    Terrain_ptr Original_Ground[NUM_ORIGINAL_GROUND];

    // the player doll. It is the referensepoint to all othet objekts in
    // the map it is not the starting possition of the player
    // the starting possition will be set by the game
    Terrain_ptr player_doll;

    // the size of the map
    int Terrain_Size[3];

    // ponter to the same data in Level
    Enemy_ptr *Level_Fiender;
    // enemies local to the Sublevel
    Enemy_ptr *Sublevel_Fiender;
    int antal_sublevel_fiender;
    // the original enemies that will be used to clone all other terrain
    Enemy_ptr Original_Enemies[NUM_ORIGINAL_ENEMYS];

    // events on the map
    Terrain_ptr Original_Events[NUM_EVENT_TYPES];
    Terrain_ptr *Sublevel_Events;

```

```

    int antal_sublevel_handelser;

    int Level_To_Play;

    // pointer to the same data in game
    Player_ptr Player_ONE;

// SUBLEVEL METHODS //////////////////////////////////////

    void Load_Terrain_Graphics(void);
    void Load_Map_Data(char filename[]);

    void Move(int x,int y);

    void Draw(void);

public:
    Sublevel(char level_filename[],Player_ptr Player_source);
    ~Sublevel(void);
    int Playing(void);

} *Sublevel_ptr;

```

E.6 Level

```

// LEVEL CLASS //////////////////////////////////////
typedef class Level
{
protected:

    // the number of maps in level
    int maps;
    // the sublevel that the player is playing on
    int Level_To_Play;
    // the curent map that the player i on
    int Current_Map;

    // the maps or different rooms in level
    Sublevel_ptr karta[2];

    // the list of the enemies in the level. if you leave a room the
    // enemies are still there when you come back
    Enemy_ptr Fiender;
    // pointer to the same data in game
    Player_ptr Player_ONE;

public:
    Level(Player_ptr Global_Player);
    ~Level(void);
    // the main event loop for the level
    void Playing_Level(void);

} *Level_ptr;

```

E.7 Game

```

// GAME CLASS //////////////////////////////////////
class Game
{
protected:

```

```

// the player data. Global
Player_ptr Player_ONE;

// the Levels in the game
Level_ptr Level_1;

// Animations

// GAME FUNCTIONS ////////////////////////////////////////

public:

    int Game_Init(void *parms=NULL);
    int Game_Shutdown(void *parms=NULL);
    int Game_Main(void *parms=NULL);

};

#endif // __game_h_

```


F mapeditor.h

```
/*=====
*
* Copyright (C) 2004 Martin Persson. All Rights Reserved.
*
* File:      game.h
* Content:   Main Game include file
*
*****/

// watch for multiple inclusions
#ifndef __game_h_
#define __game_h_

// INCLUDES ////////////////////////////////////////
#include "bob.h"

// DEFINES ////////////////////////////////////////

#define TITLE "Martins_Spelmotor" // Title of the window
// terrain layer type defines to keep track of what you are doing
#define BACKGROUND 0
#define GROUND 1
#define FOREGROUND 2
#define NO_BLOCK_SELECTED -1
#define PLAYER_SELECTED -2

// only used for the ground layer of terrain to indicate that pixel scan
// should be used insted of the ordinary Bob::Collision
#define TERRAIN_TYPE_NON_SQUARE 1

#define NUM_ORIGINAL_GROUND 6
#define NUM_ENEMY_TYPES 2
#define NUM_EVENT_TYPES 1

#define GROUND_TYPE 1 //
#define ENEMY_TYPE 2 // used for selection on the objects on the map
#define EVENT_TYPE 3 //

// GLOBALS ////////////////////////////////////////

// PROTOTYPES ////////////////////////////////////////

// TERRAIN CLASS ////////////////////////////////////////
typedef class Terrain:public Bob
{
protected:
    int Terrain_Attributes;
public:
    // Used to identify what type of original terrain it was cloned from
    // so the filename can be saved in the *.ban file
    int Type_Index;

    double t_x,t_y;
    double t_xv,t_yv;
    // Used to make the ForeGround and BackGround move faster or slower
    // than the player
    double SpeedFactor;
};
```

```

    int Move(void);
    void Move(int X,int Y);

    Terrain(int X, int Y,int Width, int Height,int Num_frames,int Attr,
            int Mem_flags,LPDIRECTDRAWSURFACE *destination_surface);
    ~Terrain(void);
    Terrain(Terrain *terrain_src);
} *Terrain_ptr;

// MAP EDITOR CLASS ////////////////////////////////////////
// editor class
typedef class Map_Editor
{
protected:
    // Used to point at the global DirectInput mouse
    Bob_ptr mouse;

    // This array might be converted to a linked list later on
    Terrain_ptr *Ground[3];

    // The original terrain that will be used to clone all other terrain
    Terrain_ptr Original_Ground[NUMORIGINALGROUND];

    // The player doll. It is the reference point to all othet objects
    // in the map. It is not the starting position of the player.
    // The starting possition will be set by the game
    Terrain_ptr player_doll, toolbox_doll;

    // The size of the map
    int Terrain_Size[3];

    // The enemies used to clone the enemies shown on the map
    Terrain_ptr Orginal_Fiender[NUMENEMY_TYPES];

    Terrain_ptr *Fiender;

    // The evets used to clone the events on the map
    Terrain_ptr Original_Events[NUMEVENT_TYPES];
    Terrain_ptr *Events;

    int Antal_Fiender;
    int Current_Enemy_type;
    int Current_Enemy;

    int Current_Terrain_Block;

    int Current_Layer;
    int Current_Original_Block;

    int Current_Event;
    int Current_Event_Type;
    int Antal_Handelser;

    bool show_ring;

    int objekttype;

    // methods

    // Adds an enemy to the map
    void New_Enemy(int Enemytype);

```

```

// Removes an enemy from the map
void Delete_Enemy(int index);
// Adds an event to the map
void New_Event(int index);
// Removes an event from the map
void Delete_Event(int index);
// Returns true if the block is clicked
bool Terrain_Click_Left(Terrain_ptr block);
bool Terrain_Click_Right(Terrain_ptr block);
// Method that adds an object to the map
void New_Terrain(int original_terrain_index, int layertype);
// Method that removes an object in the map
void Delete_Terrain(int index, int layertype);
// Saves the map to a *.ban file
void Save_map(char filemane[]);
// Loads a valid *.ban file into the memory
void Load_map(char Filemane[]);
// Moves a terrainblock
void Move_Terrain_Block(int index, int layertype, int x_distance,
                        int y_distance);

// Structuring of Editing()
void Mouse_Editing_stuff(void);
void Keyboard_Editing_stuff(void);

void Move_Map(int x, int y);
void Move_Map_Objekt(Terrain_ptr objekt, int x, int y);

public:
// the konstruktor
Map_Editor(Bob_ptr global_mouse);
// the destruktork
~Map_Editor(void);
// to be used in the main event loop
void Editing(void);
// method that draws all terrain blocks and enemys and events
void Draw(void);

} *Map_Editor_ptr;

// game console
class Game
{
protected:
// GAME OBJEKTS //////////////////////////////////////
Map_Editor_ptr editorn;

public:
// the mouse
Bob_ptr mouse; // public because the editor needs it

// GAME FUNCTIONS //////////////////////////////////////
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);
};

#endif // __game_h_

```

G Screenshot

Figure G.1 shows what the test game looks like. The arrows are velocity vectors and

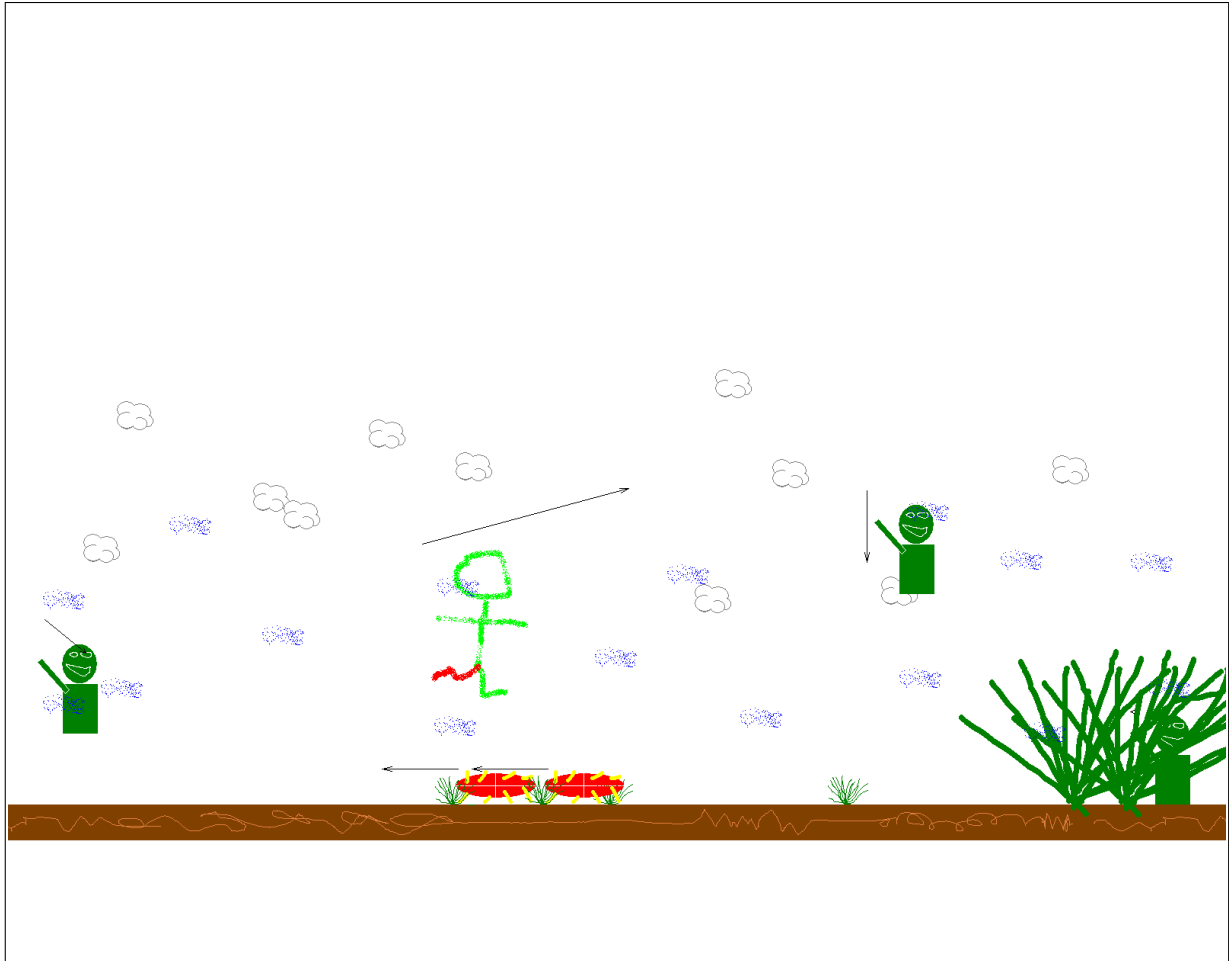


Figure G.1: A Screen Shot of an Example Game

aren't actually shown during gameplay.