University of Reading

Department of Computer Science

3/CS/6N - Computer Science Project

Final Project Report

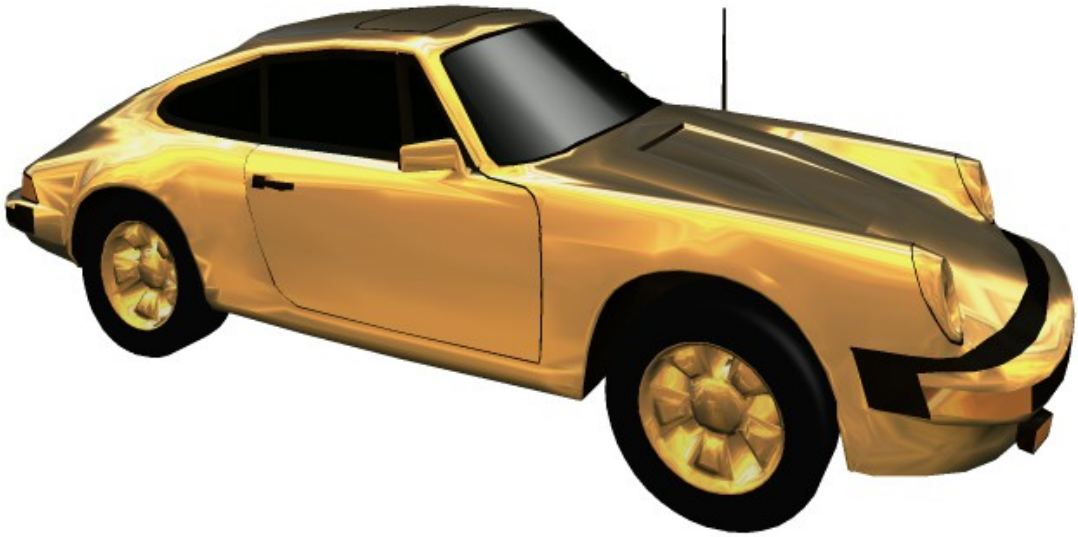
'Velocity' 3D graphics engine

By Nicholas Holland

Supervisor: Dr J.A.D.W. Anderson.

Second reader: Dr J.P. Thomas

5th May 99



Abstract

3D graphics are becoming an essential part of most modern computer games. They are used to add a greater sense of realism to the game playing experience. The aim of this project was to create a 3D graphics engine which, could be used to form the basis of a 3D game. More specifically a 3D driving simulation.

In order for a game to be playable, the graphics on the screen have to be updated many times a second. An acceptable screen update is between 15 and 70 frames per second. This constraint means that the 3D engine has to be able render as much as possible within a short period of time. For this reason, it was decided that the 3D engine would be programmed in Assembler for the Intel Pentium processor.

The project has used the prototype life cycle model. The prototype version of the 3D-engine has been written in TopSpeed Modula-2. Due to the memory restrictions imposed by Modula-2, some of the features found in the final system are not present in the prototype.

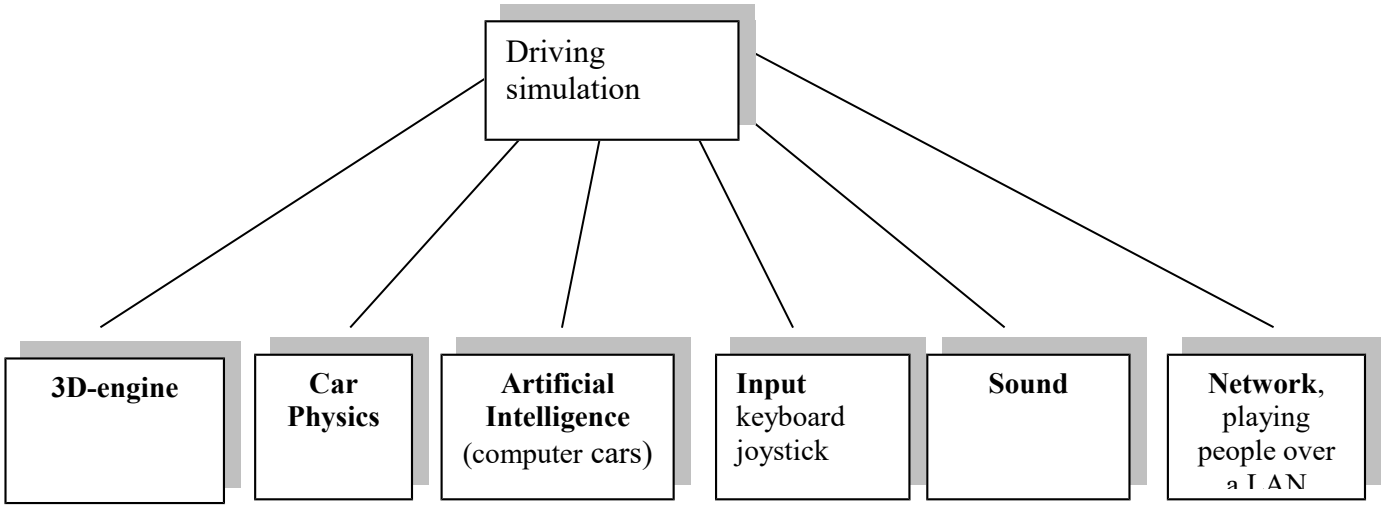
The final system was originally designed to be implemented entirely in Assembly language, unfortunately due to the complex nature of 3D graphics and time constraints it has only been possible to implement the final system in Microsoft Visual C++ with some inline assembler. If the project were to be continued, the aim would be to convert almost all the C++ code in to assembler to increase execution speed.

Currently the final system is fully functional and working. It runs under Microsoft Windows 95 / 98, as a Windows application. The 3D-engine has been configured for use in two screen modes. The first mode works at a low screen resolution capable of rendering images at a rate of up to 70 frames per second. The second mode works at a high screen resolution rendering images at a rate of around 1 frame every five seconds. Through out this report all the rendered images have been done using the Velocity 3D engine in the high-resolution mode.

2. Introduction to project and Specification

Introduction

To understand how a 3D-engine fits in to a game such as a driving simulation it is useful to take an overview of what components make such a game. The diagram below shows the main components required to create a modern driving simulation for the PC.



One of the most important components in the diagram is the 3D-engine. The 3D-engine affects how the graphics in the simulator look and also what frame rate the game runs at. The graphics of a game can determine whether it will be a commercial success or failure. The first thing people look at when buying a new game is the graphics. If the game has unrealistic slow graphics, then the game will be less appealing compared to a game with fast realistic graphics. Other factors such as car physics and artificial intelligence are considered to be of less importance in comparison with the 3D-engine.

When developing a 3D game, most of the time and resources will be spent on designing and implementing a 3D-engine because of the factors mention above. The 3D-engine is the most complex and critical component of the game.

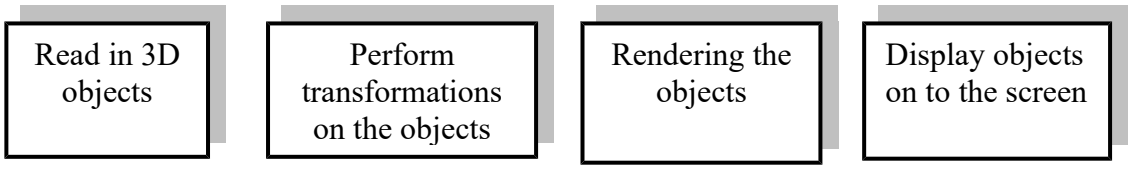
Over the past five years games programmers have worked on developing highly optimised 3D-engines with reasonable success. Despite this, the 3D-engine still remains the most demanding component on the processor. For example, when executing a 3D-game, around 50% of the processing time will be spent performing calculations for the 3D-engine every frame. This leaves the other 50% of processing time to perform all the other tasks. For this reason two changes are taking place in the computer industry.

The first has been the introduction of graphics cards which, have built in processors for rendering 3D-graphics. This allows for functions normally done in software by the processor to be done in hardware by the graphics card. The result of which is that the 3D-engine requires less processor time, therefore increasing the frame rate.

The second change is in the processor market. Companies such as Intel, AMD and Cyrix have started to include extra functions in their processor instruction sets to speed up the operations required to process 3D graphics. For example the chip manufacturer AMD [3] has produced a processor called the K6, 3D NOW. The AMD processor contains extra instructions to perform hardware matrix operations. The extra instructions are aimed specifically at 3D-games.

Aims and Objectives

The aim of the project is to write a 3D graphics engine in assembler for a driving simulation. A 3D-engine can be broken down into four distinct sections as shown in this diagram.



The objectives of the project were to design and implement all of the above sections and provide an application programming interface (API) to them. The API should allow the 3D-engine to be integrated into a game with the minimum amount of effort. The API should be abstract enough to allow a person with average 3D knowledge to be able to use it.

For the 3D-engine to be useable it has to be a fast. The aim of the project was to get the 3D engine, running on a 300Mhz Pentium II, to draw at least 50,000 texture mapped triangles to the screen every second. This means around 1600 triangles per frame, when running at 30 frames per second.

Another objective is to make the 3D-engine as compatible and reliable as possible. 3D-engines perform a lot of mathematical calculations, there are a number of situations where problems such as overflows, division by zero and memory leaks can occur. This means extensive testing needs carrying out to make it as reliable as possible. The 3D-engine should be compatible with all Pentium PC's given that they meet the minimum requirements of the 3D engine.

Review of published work

The 3D graphics engine is extremely important to the success of a game. It controls both the games visual appearance and how fast the game runs. These factors partly depend on which techniques are used to manipulate and display 3D objects. This is a list of various techniques that are used in a collection of the latest driving simulations for the PC:

Technique used	Screamer 2[1]	Motor Head [5]	Grand Prix Legends [6]	Colin McRae Rally [7]
Flat shading	Yes	No	No	No
Gouraud shading	No	Yes	Yes	Yes
Texture mapping	Yes	Yes	Yes	Yes
Environment mapping	Yes	No	No	Yes
Depth buffering	Yes	Yes	Yes	Yes

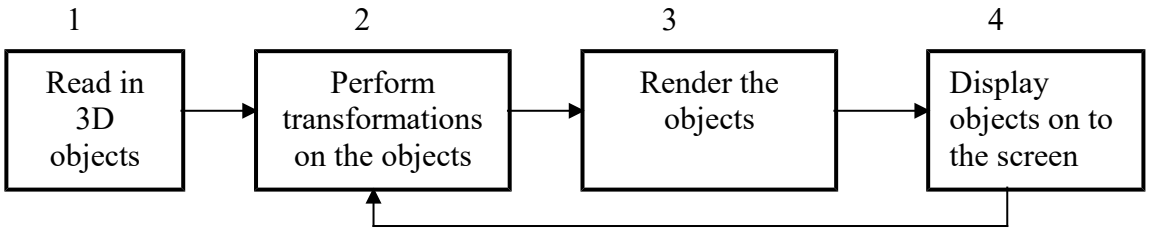
From the list, it was decided that the 3D engine would include flat shading, texture mapping, Gouraud shading, environment mapping and depth buffering. The 3D-engine would also include phong shading. The different techniques relate to how objects are displayed on screen and what appearance they take on. In general, the more features a 3D-engine the more realistic the graphics are.

From the review of the listed games it was also decided that the minimum requirements for a PC to run the 3D-engine would be a Pentium 166, with 32MB of RAM. This is the minimum requirement for all four of the games listed.

General statement of problem

"The term 'engine' is used to describe a piece of software that accepts data and acts on that data. Three dimensional graphics engines use geometrical descriptions of objects to produce rendered images on screen." John De Goes, game devleoper[2]

In order to understand the problem it is necessary to break it down into the following components:



The first stage involves reading in geometric representations of 3D objects from a file. The design issues faced here are determining what type of file format to use. Whether to use an existing file format, or create one. The format of the data structures used to store the geometric data need to be determined. The second stage involves performing mathematical transformations on the 3D data. For example, rotate object by 30 degrees to the left.

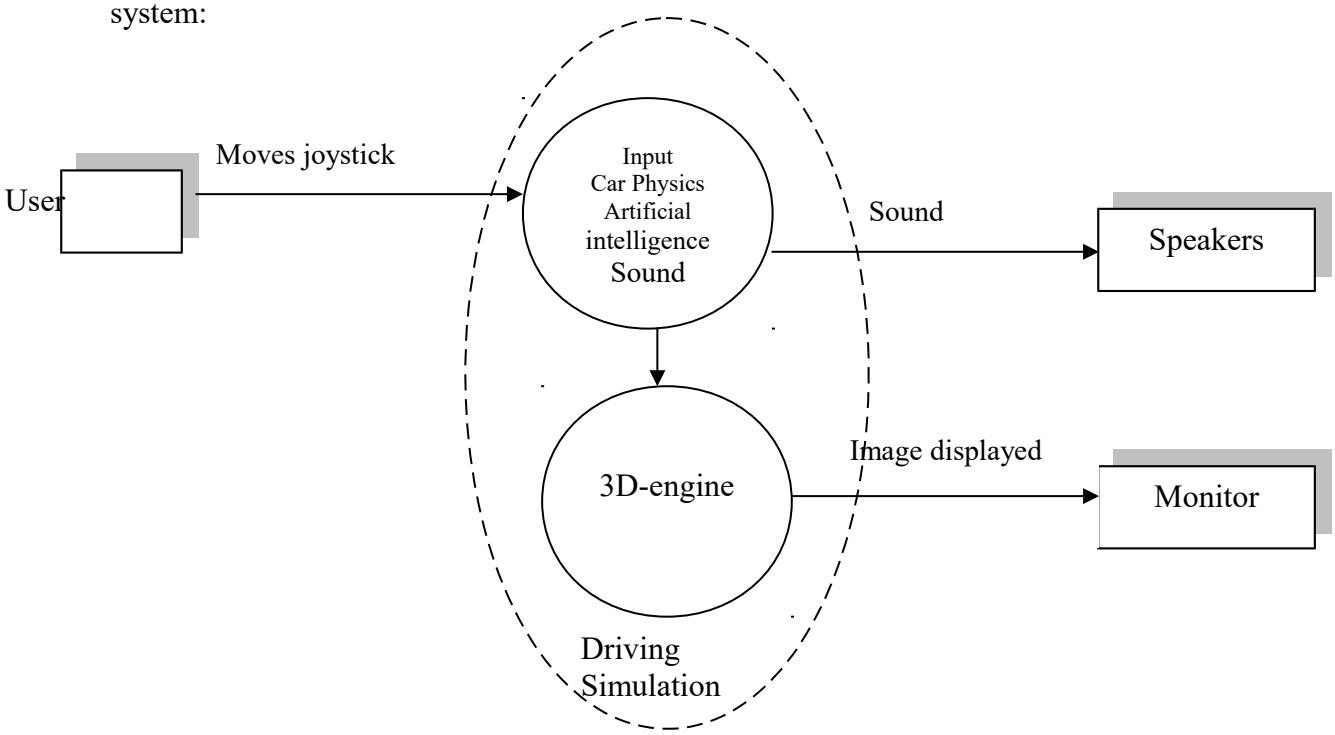
The third stage involves taking the transformed 3D data and rendering an image in memory. The final stage involves initialising the graphics card in to a desired screen resolution, then copying the rendered image from memory into the graphics card.

The main problem faced when writing a 3D-engine is performance. Stages 2,3 and 4 will execute every frame. In order to keep the frame-rate high, each stage has to execute in a minimum amount of time. This means when implementing the 3D-engine the code must be optimised as much as possible. This is the main reason for deciding to implement the 3D-engine in Assembler.

3. Analysis and Design

Overview of the system

The purpose of the 3D-engine is to allow a programmer who is writing a driving simulation to import it as a library, then use its functions to do all the 3D graphics for the simulation. This diagram represents an example of the overall data flow in such a system:



This project is only concerned with the flow of data between the 3D-engine and the monitor. To understand what the 3D-engine should do, here is a list of the functional and non-functional requirements.

Functional Requirements

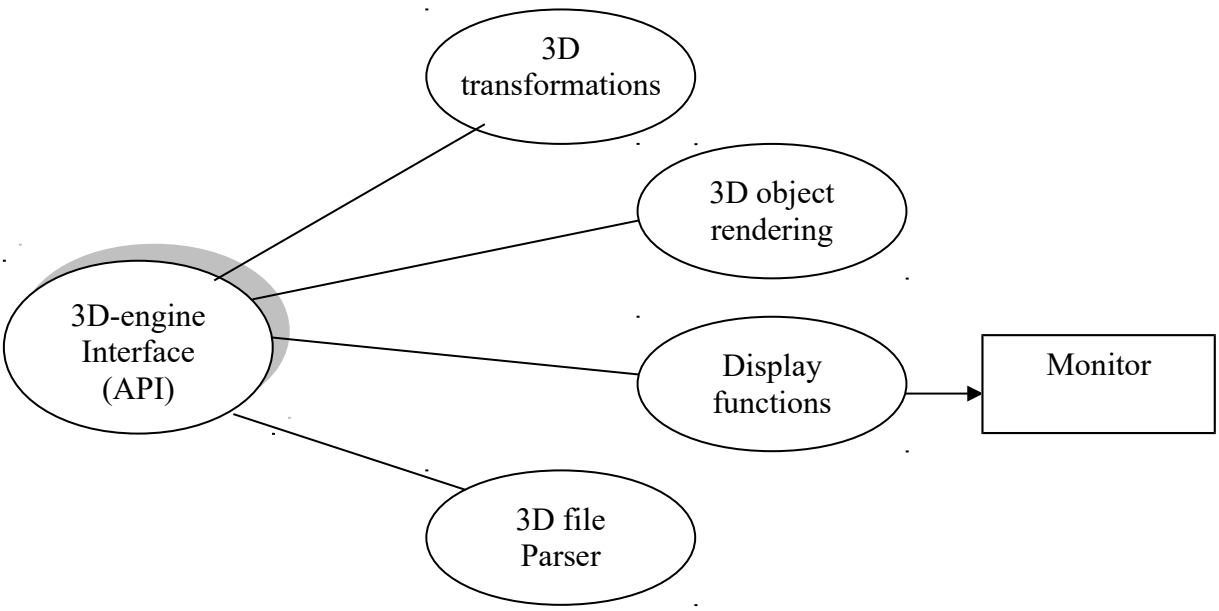
- Capable of reading files from a disk containing 3D geometric representations of objects, then loading them into memory.
- Capable of performing 3D transformations on 3D objects.
- Capable of rendering the 3D objects, to be displayed on a monitor.
- Capable of interfacing with the graphics card to set various screen resolutions.
- Capable of using double buffering to produce smooth animation, without any screen tears.

Non Functional Requirements

- Must be able to execute on a Pentium P166 PC running Windows 95/98 with 32Mb of RAM.
- When running the 3D-engine on a 300Mhz Pentium II it must be able to draw at least 50,000 texture mapped triangles to the screen every second. This means around 1600 triangles per frame, when running at 30 frames per second.

Detailed analysis

To understand exactly what the 3D-engine should do, it needs to be broken down into its various component parts. Here is a more detailed view of the 3D-engine.



The above diagram is designed to give an outline of how the main components of the 3D-engine are linked together. The 3D-engine interface has control of all the other components. The interface should be designed so that the programmer of the driving simulation does not need to be aware, or have knowledge of the other components of the 3D-engine. It should be as abstract as possible.

The following pages perform a detailed analysis on each of the components in the diagram.

3D-engine Interface

The 3D-engine interface has to provide a complete set of functions for generating 3D-graphics. This list outlines the main the functions required.

- Ability to load a new object. For example, `LoadCar("car3.dat");`
- Ability to scale, move and rotate an object in 3D space.
- Ability to draw an object to the screen
- Ability to set the screen resolution on the graphics card.
- Ability to allocate and de-allocate memory for the objects.

The aim of the functions is to make generating 3D-graphics as logical and simple as possible.

3D file Parser

The file parser must be able to load files containing 3D data. The parser loads the 3D information and stores it in memory. This component therefore only needs to supply one function to the 3D-engine interface, that is `LoadCar`.

When creating a driving simulator there needs to be a way of modelling and generating objects, such as cars. Rather than creating a modeller specifically for the 3D-engine, it is more practical to use an existing off the shelf modeller. Therefore when the modeller is chosen, the parser must be able to read the files exported from it.

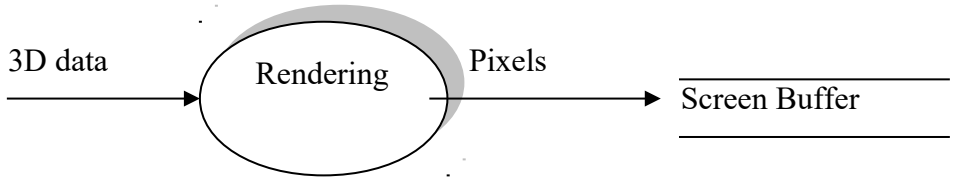
3D Transformations

The 3D-transformations component should provide a number of functions for manipulating 3D objects. This list contains the functional requirements.

- Must provide a function for moving objects in 3D-space using rotation and translations.
- Must provide functions for manipulating a camera, through which the objects are rendered.
- Must provide a function called '`DrawCar`' for drawing 3D-objects. This function acts only as an interface to the 3D object rendering component.

3D object rendering

After a 3D object has undergone 3D transformation it then needs rendering.
The rendering phase is the stage where the object is actually drawn.



As the objects are rendered the pixels generated are stored in a screen buffer. If it the pixels were to be displayed straight to the graphics card then it would be possible to see the objects being drawn, an unwanted side-effect. The rendering stage is possibly the most complex part of the 3D-engine. The following functional requirements are designed to give an overview of what the renderer needs to do. The techniques listed will be explained in full during the design phase.

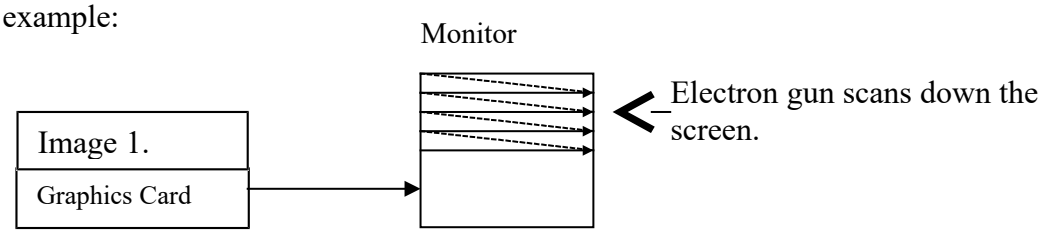
Functional Requirements:

- Capable of drawing polygons
- Capable of performing the following shading algorithms: Flat, Gouraud and phong shading.
- Capable of performing texture mapping and environment mapping.
- Must use a depth-buffer.
- Must use a screen buffer to draw the rendered image to.

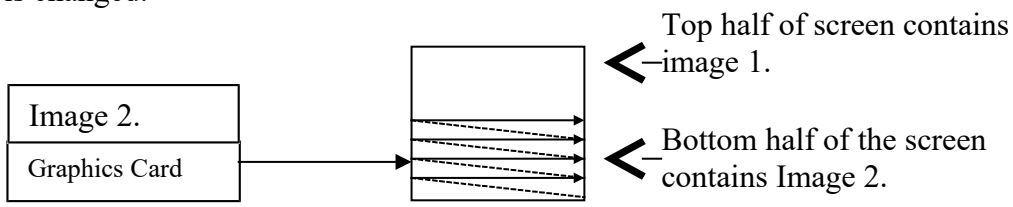
Display functions

The display functions should be an interface between the 3D-engine and the graphics card. The first function that is required is to set the graphics card to a desired resolution. For example, a screen resolution 320 pixels wide by 200 pixels high with 24bit colour. The second function is to transfer the image held in the screen buffer to the graphics card. Special care needs to be taken to ensure that certain side effects such as screen tearing will not occur.

Screen tearing is a situation which, is caused by the lack of synchronisation between the monitor and the graphics card. It occurs when contents of the graphics card is updated while the electron gun in the monitor is scanning down the screen. The picture on the monitor will have a different image on the top half to the bottom half. For example:



Before the electron gun gets to the bottom of the screen the image held in the graphics card is changed.



The overall affect is that the screen appears to look disjointed.
Here is a list of the functional and non-functional requirements for the display functions:

Functional Requirements:

- Capable of setting various screen resolutions, ranging from 320x200 to 800x600
- Capable of copying the screen buffer into the graphics card and displaying it on screen.

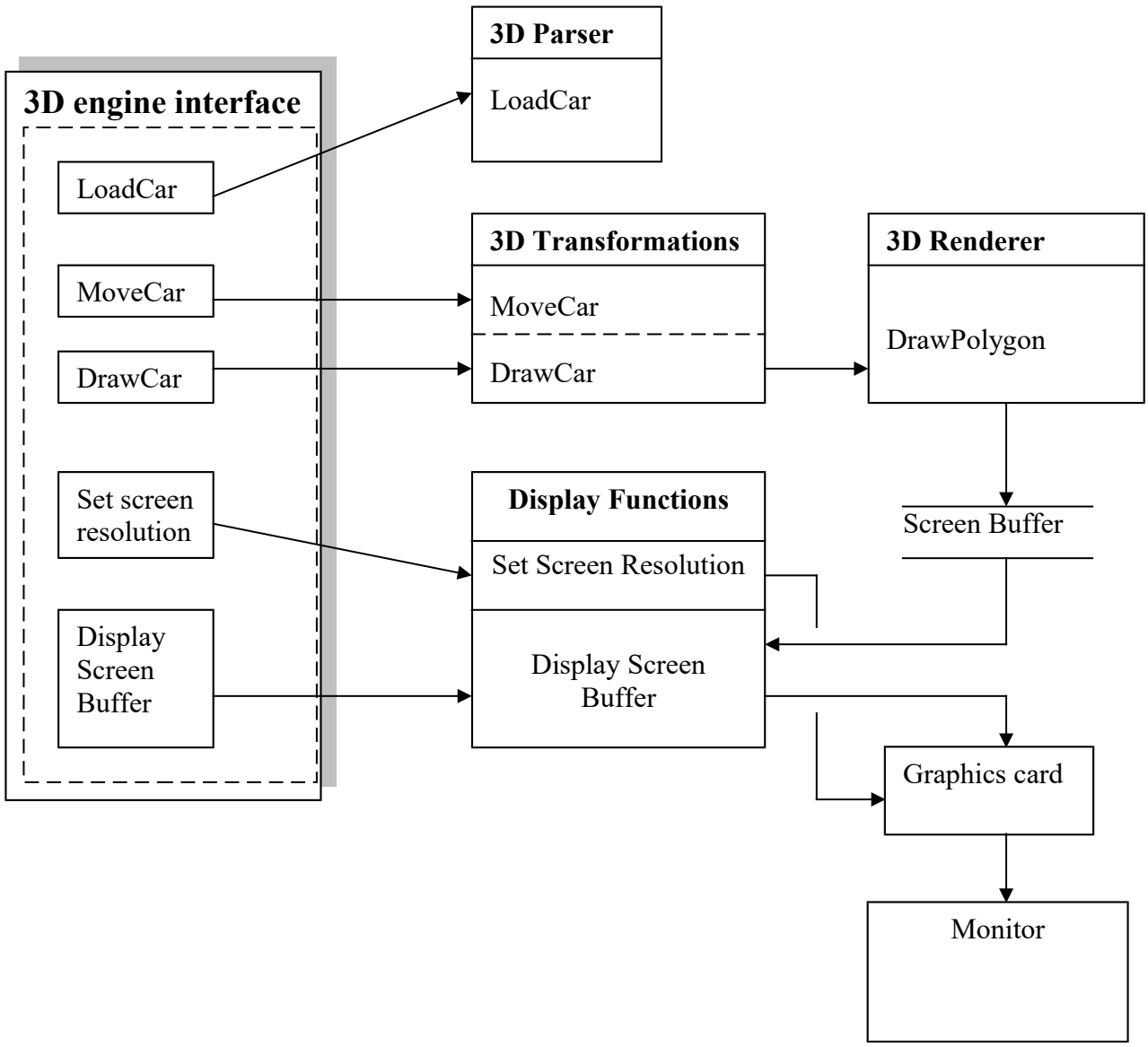
Non Functional Requirements:

- Should avoid screen tearing
- Must be compatible with most modern graphics cards manufactured within the last two years.

Design

The previous section on analysis was concerned with *what* the 3D-engine must do, this next section on design is concerned with *how* the 3D-engine should work.

Taking into account the various functional requirements the next diagram gives an overview of how the various components should be linked together.



The following pages will look at the design of each component in more detail.

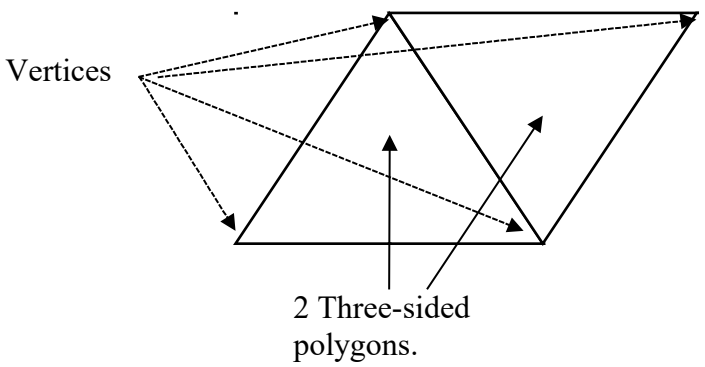
Data Design

Before being able to design the components that make a 3D-engine it is important to design the data-types that will store the information. The design of the data structures is a key element to how the 3D engine will function. To define the data-types it is necessary to determine what actually makes a 3D object. Here is a quote from Josh White, an expert in 3D modelling:

"Real-time 3D graphics are composed of points in space connected by simple surfaces. The surfaces are always flat with straight edges (though they can appear curved if we use lots of small ones). These surfaces are called polygons (means "many sides"). The points are called "vertices".

A very astute observer would take these definitions and realise that the world of real-time 3D is composed of infinitely thin faces, like a paper shell, instead of solid objects. Also, these thin faces are like one-way mirrors; they're invisible from one side. "

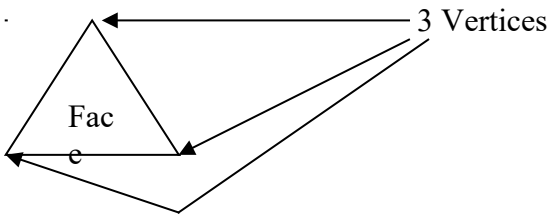
Therefore, an object is made up of vertices, which are connected together by polygons. Here is an example of a very simple two polygon object.



The first major design issue that needs resolving is to decide what type of polygons the 3D-engine should use. Some 3D-engines can draw n-sided polygons, while others draw just three sided or four sided polygons.

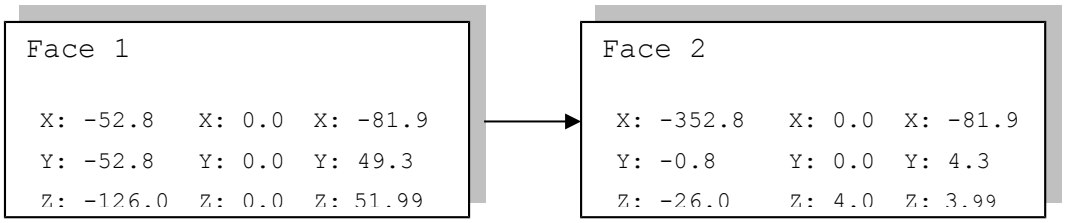
After performing research into the area, it was decided that the 3D-engine would use 3 sided polygons (triangles). The algorithms for drawing triangles are generally faster and easier to implement than for 4 sided or n-sided polygons. This is probably why most 3D-games for the PC use triangles to draw objects as standard. This factor has become even more evident in the past two years with the introduction of PC graphics cards which have hardware acceleration specifically designed to draw triangles.

A single triangle consists of an infinitely thin face and 3 vertices. The triangle is only visible from one side.

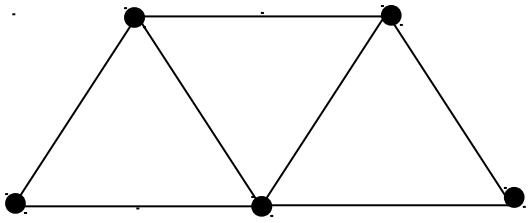


3D object design

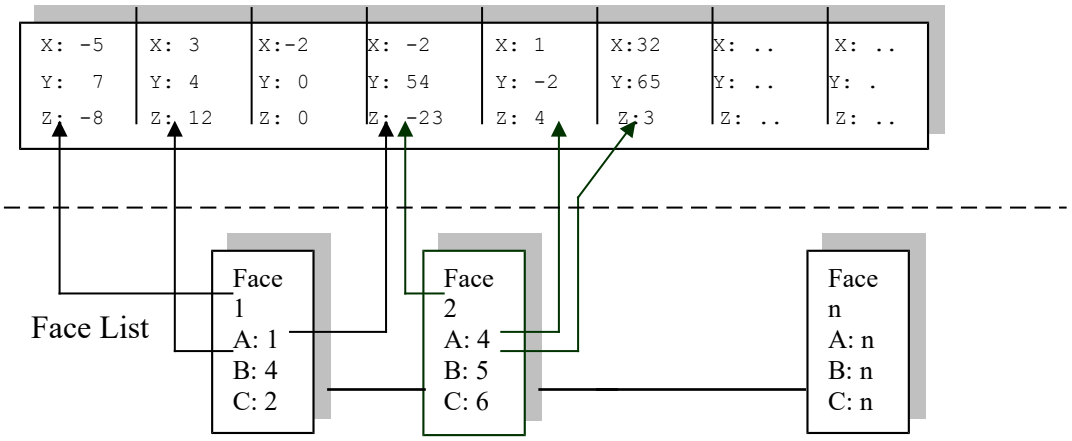
One possible way of storing an object would be to have a list of faces. Each face is then assigned three vertices, which define its structure. For example



This method will work, but is not particularly efficient. Rather than storing three vertices for each face, it is more efficient to use indexes to a list of vertices. For example, in the object below, if each face stored three vertices then a total of nine vertices will be stored for the whole object. However if each face used an index to a list of vertices then only five vertices would need storing for the object (vertex sharing). This factor becomes very significant with large objects.



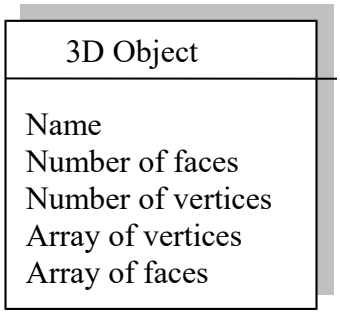
Vertex list



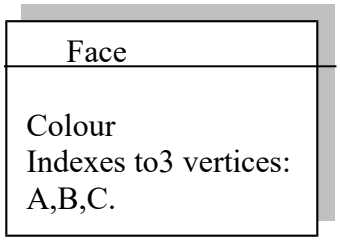
The above diagram shows how faces share vertices with other faces, A,B and C are indexes to the vertex list.

3D data structures

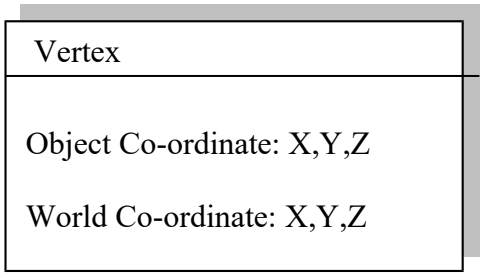
Using the information about vertices and faces it is possible to construct the data structure for a 3D object.



A face can be defined as follows:



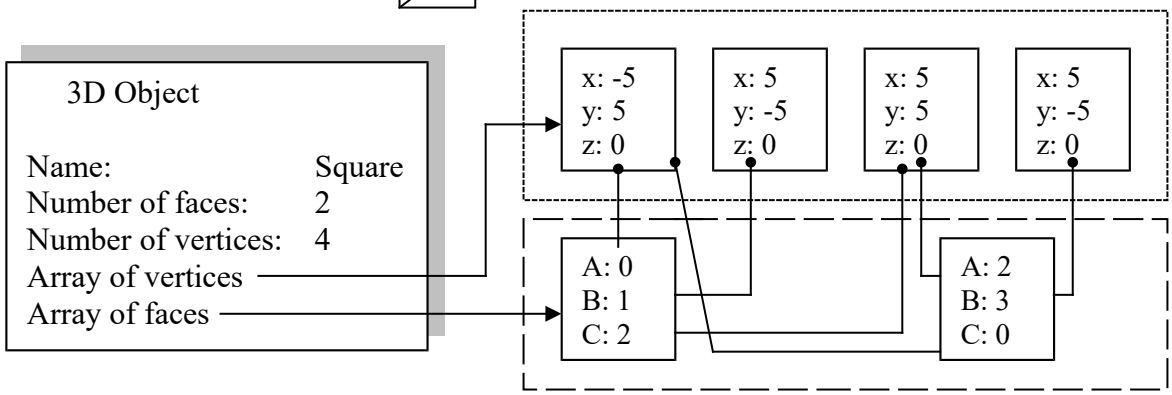
The data structure for a single vertex is as follows:



A vertex contains two elements. One element is the 'Object Co-ordinate' which represents the position of the vertex before any transformations have taken place. The 'Object Co-ordinate' is fixed to the value set by the parser when reading in a 3D object from a file. The second element is called the 'World co-ordinate'. At the beginning of each frame, the 'Object Co-ordinate' is copied into the 'World Co-ordinate'. All transformations then take place on the 'World Co-ordinate'. It is effectively a temporary variable.

Data-Structure Overview

To give an overview of the data structure, here is a simple example of a square represented as a 3D object. 



Parser Design

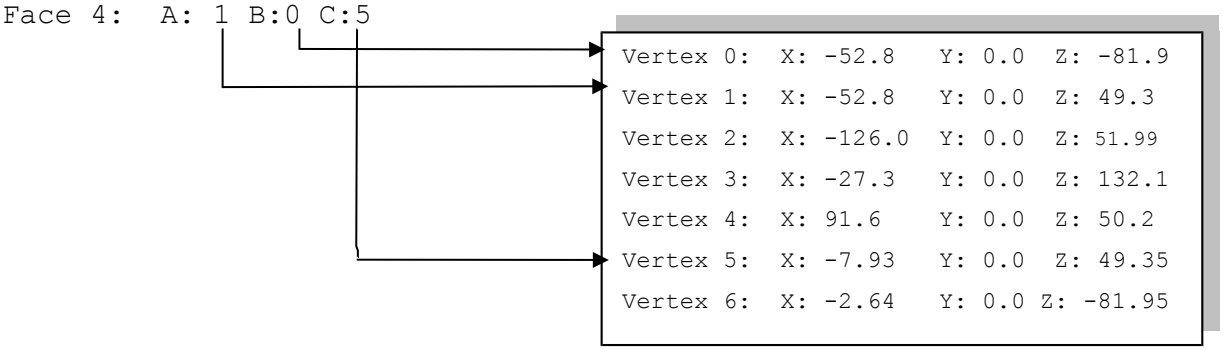
After designing the data-structure for a 3D-object, the next logical stage is to design the parser to fill the 3D-object with information.

The 3D-modeller selected to create 3D object files for the parser is 3D Studio by AutoDesk. 3D Studio is a modelling tool that has almost become a computer games industry standard. Companies such as Lucas Arts (Tie Fighter), WestWood Studios (Command and Conquer), and Maxis (Sim City 2000) all use 3D Studio.

3D studio has the ability to export 3D objects in a text-based format called ASC. The layout of the ASC format is as follows:

```
Named object: "Object01"
Tri-mesh, Vertices: 7     Faces: 5
Vertex list:
Vertex 0:  X: -52.875084     Y: -0.000018    Z: -81.956375
Vertex 1:  X: -52.875084     Y: 0.000004     Z: 49.350079
Vertex 2:  X: -126.018944    Y: 0.000004     Z: 51.993832
Vertex 3:  X: -27.318792     Y: 0.000018     Z: 132.187698
Vertex 4:  X: 91.650139      Y: 0.000004     Z: 50.231327
Vertex 5:  X: -7.931263      Y: 0.000004     Z: 49.350079
Vertex 6:  X: -2.643754      Y: -0.000018    Z: -81.956375
Face list:
Face 0:    A:5 B:4 C:3
Face 1:    A:3 B:2 C:1
Face 2:    A:5 B:3 C:1
Face 3:    A:0 B:6 C:5
Face 4:    A:1 B:0 C:5
```

The first line contains the object name. The second line contains the word `Tri Mesh`. This is short for triangle mesh and means that the object is made out of triangles. 3D studio uses triangles as standard. The remainder of the line contains '`Vertices: 7 Faces: 5`'. This represents that the object consists of 7 vertices and 5 faces. Following that is a vertex list containing all the vertices used by the object. The second section is the face list. Each face has a list of three variables A,B,C.
Each of these variables relates to a vertex, in the vertex list. For example, face 4 has the following points



The layout of the ASC file format is very similar to the structure of a 3D-object. Therefore parsing a ASC file should be fairly efficient. Here is a C prototype for LoadCar function within the parser.

```
int LoadCar(char *FileName, 3D-Object *Car)
```

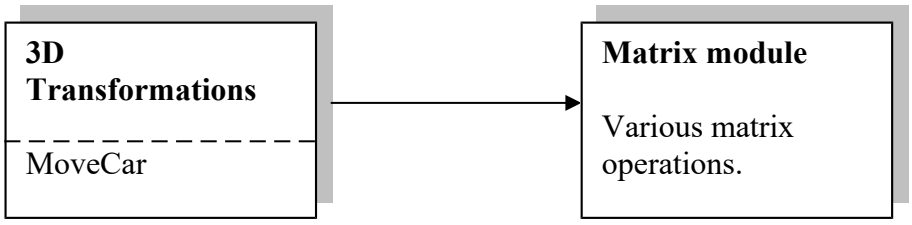
The function is passed a filename which, should refer to an ASC file in the current directory. If the file is not found the function should return an error value to inform the calling program. If the file exists, it should be opened. The parser should then read the file and start adding data to the 3D-object. The 3D-object is passed to the function as a pointer. If the file has been parsed successfully then a return value indicates this to the calling program.

3D-transformations design

From the specification, the 3D-transformation component has to provide functions to do the following

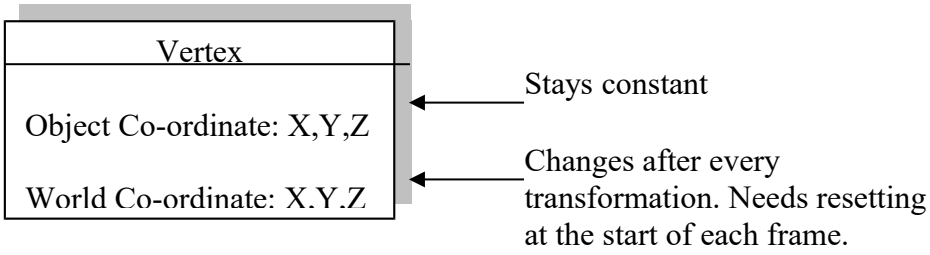
1. Object Scaling
2. Object rotation and translation
3. Camera manipulation
4. DrawCar function

In order to perform geometric operations on 3D graphics it is almost essential to use matrices. Therefore the inclusion of a separate module specifically for performing matrix operations is necessary.



Using matrices it is possible to scale, rotate and translate all the points in a 3D-object.

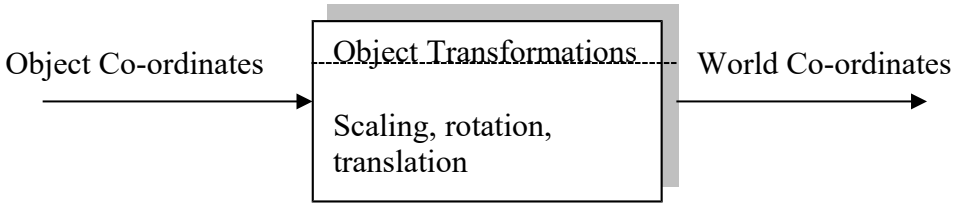
During the transformations that take place, the 3D-object has its 'World Co-ordinates' altered every frame. At the beginning of each frame the 'World Co-ordinates' need to be reset, to equal the original 'Object Co-ordinates'. If this did not happen the 3D-engine would not work. For example, if a car is to be rotated 45 degrees one frame, then 46 degrees the next and the co-ordinates were not reset, the car would end up turning 91 degrees instead of the desired 46 degrees.



Here is an outline for the design of a reset function.

```
void Reset_Car(3D-Object *Car);
for n=0 to Number_of_Verticies do

VertexArray[n].World_Coordinate=VertexArray[n].Object_Coordinate

end;
```

After resetting the 'World Co-ordinates' of an object, the 3D-engine can start to perform some transformations. The three transformations that can be performed are scaling, rotation and translation. This diagram represents the transition that takes place during the transformations.



Before any transformations take place, the object is said to be in an object co-ordinate system, after the object transformations it is said to be in the world co-ordinate system.

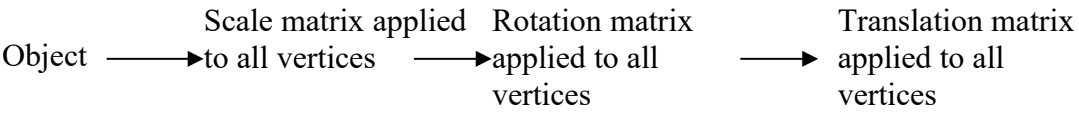
Scaling, Rotations and translations

To rotate, scale and translate objects, homogeneous co-ordinates should be used. Expressing vertices in homogeneous co-ordinates allows us to represent all geometric transformations as matrix multiplication. Each vertex in homogeneous form will have four co-ordinates, they are reduced to three dimensions by the following normalization operation.

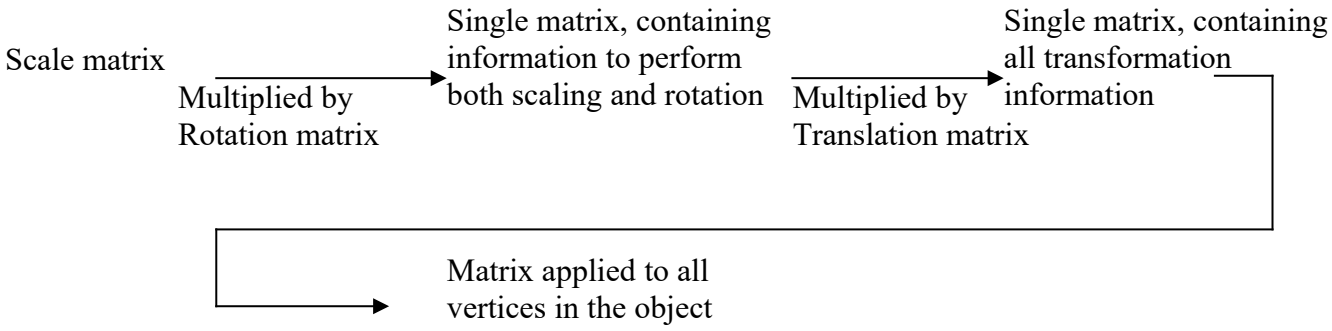
$$[x, y, z, h] \longrightarrow \left[\frac{x}{h} \quad \frac{y}{h} \quad \frac{z}{h}\right]$$

The most convenient way of performing co-ordinate transformations is to set up matrices containing the parameters of the transformations, and use matrix multiplication to reduce a series of transformations into a single transformation matrix.

Here is an example of using matrices without matrix multiplication to reduce a series of transformations into a single matrix.

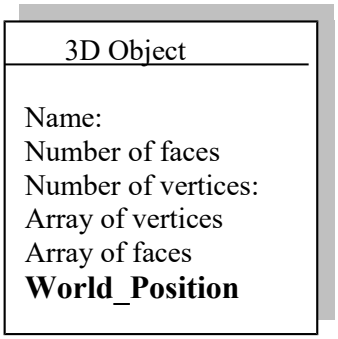


Each vertex in the object is multiplied by a matrix three times, once for each transformation. Here is an example of using matrices with matrix multiplication to reduce a series of transformations into a single matrix.



The end result of using matrix multiplication is that each vertex in the object is only multiplied by a matrix once, considerably faster than three times as in the previous method.

Therefore, to increase efficiency, the 3D engine performs all transformations on just one matrix. The matrix is stored as part of the 3D-object and is called 'World_Position'. After all the transformations are complete the World_Position matrix is multiplied with every vertex to give their new positions.



Here is the pseudo code design for ScaleCar:

```
ScaleCar(Xfactor, Yfactor, Zfactor, 3D-Object *Car)

create_ScaleMatrix(Xfactor,Yfactor,Zfactor,TempMatrix);
MatrixMultiply(TempMatrix,Car.World_Position);

end;
```

The function MoveCar, performs both rotation and translation. Here is the pseudo code design for it:

```
MoveCar(Xrot,Yrot,Zrot,Xdistance,Ydistanace,Zdistance,3D-Object *Car)

create_RotationMatrix(Xrot,Yrot,Zrot,RotationMatrix);
MatrixMultiply(RotationMatrix,Car.World_Position);

create_TranslationMatrix(Xdistance,Ydistance,Zdistance,TransMatrix);
MatrixMultiply(TransMatrix,Car.World_Position);
end;
```

Camera Co-ordinates

Before a frame can be draw, the 3D-object must be transformed from the world co-ordinate system into a camera co-ordinate system. This is required because we need to know where the object is in relation to the camera.

To define the camera co-ordinate system a function needs designing to position a camera withing the world, and to define its orientation. The data-structure for a camera is defined as follows:

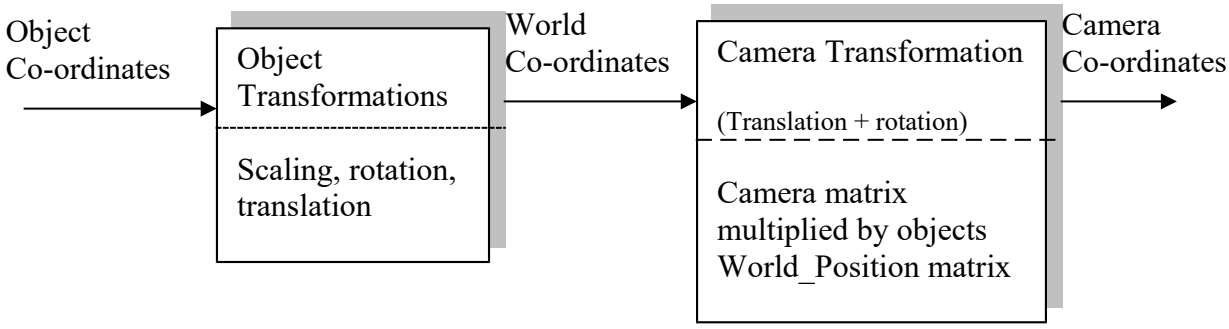
```
TYPE Camera = RECORD

                 Camera_Position
                 Roll,Pitch,Swivel
                 Focal_Length
                 Camera_Trans:Matrix;

              END;
```

The Camera_Position specifies the position of the camera in 3D space. Roll, Pitch and Swivel represent the cameras orientation, i.e. which way is it pointing. The Focal_Length defines the focal length of the camera, adjusting this will adjust the field of view. The Camera_Trans variable is similar to a 3D-objects World_Position matrix except it is inverse, e.g. a translation followed by rotation.

Here is the pseudo code for setting the camera position:

```
Set_Camera_Position(Xpos,Ypos,Zpos,Focal,Xrot,Yrot,Zrot,Camera *Cam)

              Camera_Position.x:=Xpos;
              Camera_Position.y:=Ypos;
              Camera_Position.z:=Zpos;
              Pitch:=Xrot;
              Swivel:=Yrot;
              Roll:=Zrot;
              Focal_Length:=Focal;
              SetIdentity(World);

            CreateInvMoveMatrix(Xrot,Yrot,Zrot,Xpos,Ypos,Zpos,World);

             Camera_Trans:=World;

end;
```

To convert from the world co-ordinate system to the camera co-ordinate system, the world co-ordinate system is translated so that the camera view point is at the origin. It is then rotated so that the camera is looking down the z-axis. When the function SetCameraPosition is called, the matrix required to convert the world co-ordinate system into the camera co-ordinate system is calculated. The matrix is stored in the cameras data structure.



This is the pseudo code for the design of function to perform the world to camera co-ordinate transformation.

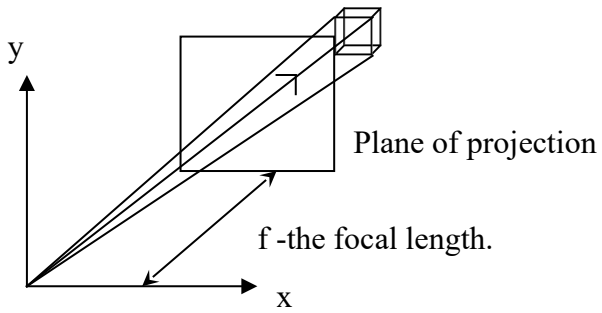
TransformCar(CarObject *Car; Camera *Cam);

```
                MatrixMultiply(Cam.Camera_Trans,Car.World_Position);
                ApplyObject(Vertices,No_Vertices,Car.World_Position);

end;
```

Because this is the last matrix transformation the object has to go through, the final World_Position matrix is applied to all the vertices of the object.

Perspective projection

Once the world co-ordinate descriptions of the objects in a scene are converted to camera co-ordinates, the 3D-objects need projecting onto a two dimensional image plane.



The formula for performing perspective projection on a vertex is:

$$x' = \frac{f \times x}{z} \qquad y' = \frac{f \times y}{z}$$

Where f represents the focal length of the camera. It is possible to use a perspective transformation matrix and then normalises the four dimensional vector to obtain the three-dimensional co-ordinates, to obtain the same results. However, normalising requires three divisions, for each vertex, in comparison with two for just using the formula stated. On a Pentium processor floating point divides can take up to three times longer than floating point multiplication, they should be avoided if possible. Therefore to speed up the above equations they can be replaced by

$$\frac{f}{z}$$

.

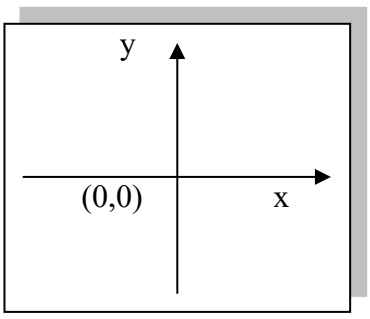
$$temp = ¿\frac{¿}{¿}\ ; \quad x' = temp \times x \qquad y' = temp \times y¿$$

This is the pseudo code design for the perspective projection function

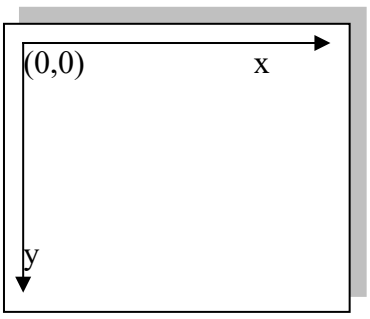
```
Camera_Projection(CarObject *Car;Camera *Cam);

for n=0 to No_Verticies
       Temp=Cam.Focal_Length/ Car.VertexList[n].z;
       Car.VertexList[n].x= Car.VertexList[n].x*Temp;
       Car.VertexList[n].y= Car.VertexList[n].y*Temp;
end;
```

Screen Transformation

The final stage of the object transformation, is to convert the project screen co-ordinates into device co-ordinates. In a 3D world, the origin should be in the centre of the screen. However, the screen co-ordinates are displayed differently, for example:



2D projected co-ordinates



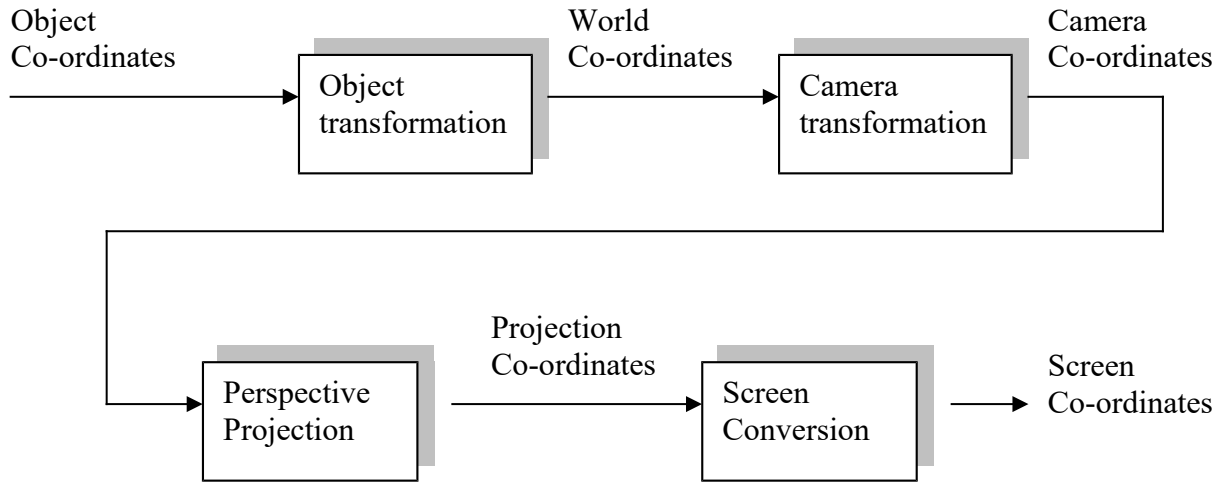
Screen co-ordinates

To convert the 2D project co-ordinates this formula can be used.

```
x=x+ ScreenWidth / 2;
y=(ScreenHeight/2) - y;
```

Overview of Object transformations

To understand the whole process that an object goes through before it is even displayed can be involving. The following diagram shows an overview of all the transformations that an object goes through. The diagram can be described as a general 3D transformations pipeline.



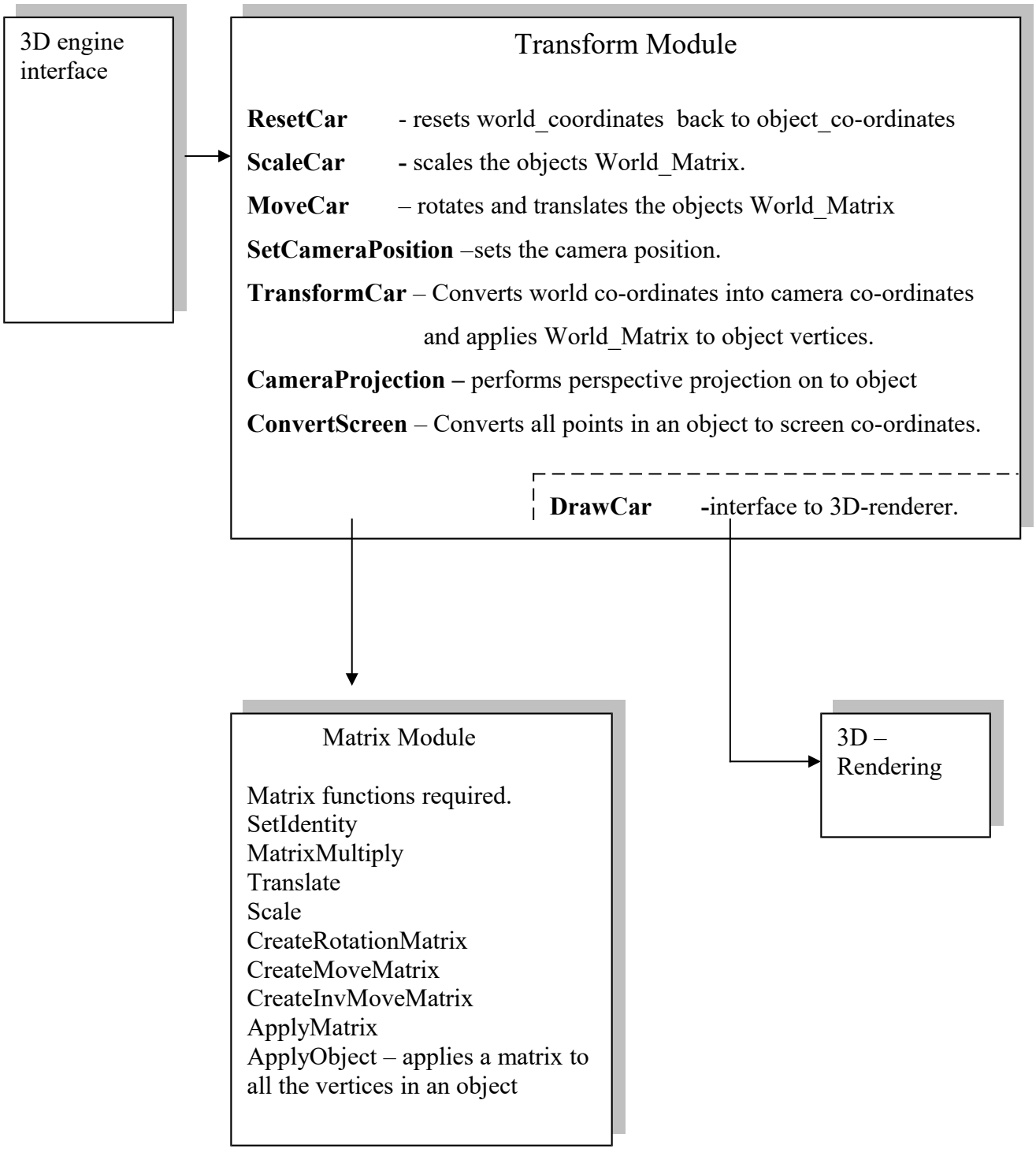
DrawCar

The last function in the transformations component is DrawCar. DrawCar is an interface to the 3D-renderer. The 3D-renderer is not aware of the 3D-object data structure, it is only aware of faces and vertices. The aim is to make it as modular as possible. The function has to perform two operations.

The first operation is to perform back-face culling to prepare the object for rendering. As already discussed triangles are only single sided. Back face culling provides a method of determining whether the triangle face is visible or not. It involves calculating the normal of the triangle to determine which way it is facing. If the normal is facing away from the viewer then the triangle is not rendered. The data structure for a face, should have a flag added to represent whether the face is visible or not.

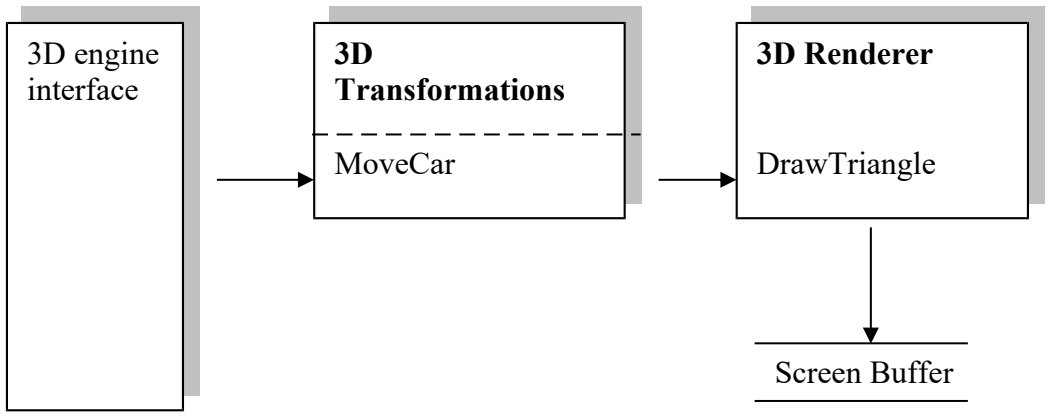
The second function it is should perform is to go through the 3D-object sending each face individually to the 3D-renderer.

Overall Design of the Transformations Component



3D Renderer

The 3D-renderer is concerned with drawing the final image. Here is a look again at how it structured within the 3D-engine.



The 3D renderer should be designed to perform the following techniques:

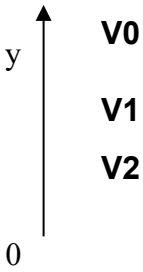
- Flat shading.
- Gouraud shading.
- Phong shading.
- Depth Buffering
- Texture mapping.
- Environment mapping.
- Bi-linear interpolation.
- Three times over sampling.

The following pages take a look at the design of each of the above techniques. The design stage starts with looking at how to draw a basic triangle.

Basic Triangle drawing

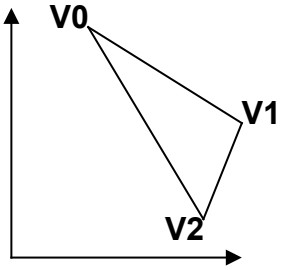
The method of drawing triangles that should be used is a simple and fast scan conversion technique. It will only work with triangles, it is not a general polygon drawing algorithm.

A triangle consists of three co-ordinates. The first part of the algorithm sorts the three co-ordinates in order of their y value. For doing this a simple function called **SortHeight** should be implemented**.** The variable names used for the three co-ordinates are V0,V1, V2. They are arranged in the following way.

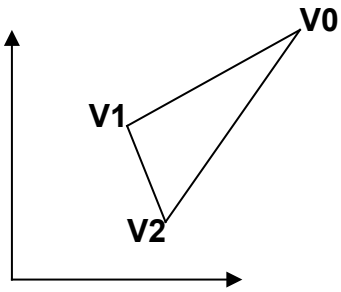


Once they have been sorted by height the algorithm calculates which side of the triangle is longest, the left side or the right side. Here is an example of the two different arrangements.

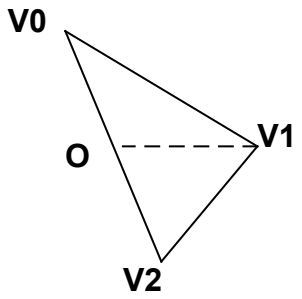
Left side longest



Right side longest



The arrangement of the co-ordinates affects how the algorithm works, which is why it is necessary to determine which side is longest. It performs the calculation by finding the co-ordinates of the variable O, shown here.

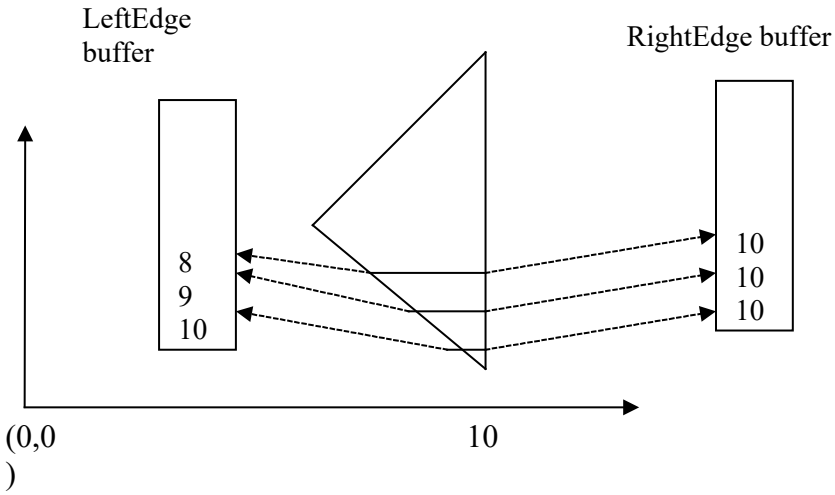


If O. x - V1. x is negative then the left side is longest, otherwise the right side is longest.

The next stage of the algorithm involves setting up two edge buffers. The edge buffers are used to scan the edges of the triangle. They are defined as follows:

```
int LeftEdge [ Screen_Height ];
int RightEdge[ Screen_Height ];
```

The algorithm then works up through the triangle, increment y by one, storing the positions of the edges in the edge buffer as it goes along. For example:



The triangle can then be displayed on to the screen using a very simple for loop.

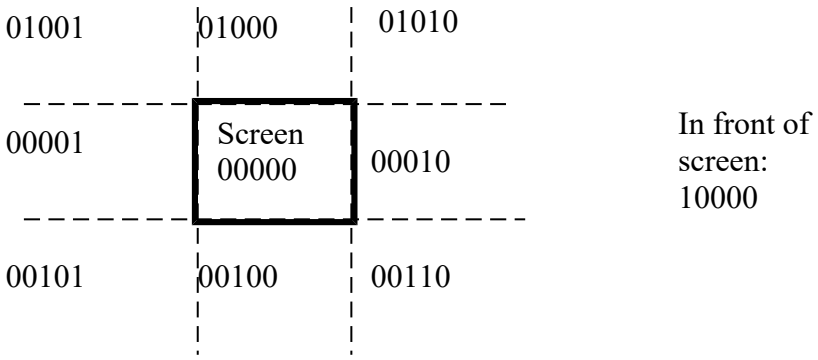
This is the C code for it:

```
for (y=V2y;y<V0y+1;y++)
       {
            SpanMin=LeftEdge[y];
            SpanMax=RightEdge[y];
            for(x=SpanMin;p<SpanMax+1;x++)
              putpixel(x,y);
      };
```

Clipping

Clipping is a way of making sure that nothing gets drawn outside the boundary of the screen. Attempting to draw just one pixel outside the limits of the screen without any control may cause the program to crash. To perform clipping it is necessary to first define the clipping rectangle. The clipping rectangle should be the width and height of the screen, however it is possible to define any region within the screen.

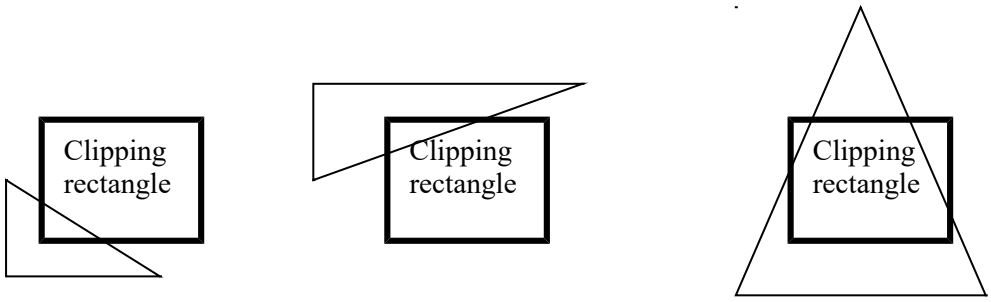
With the clipping rectangle defined it is necessary to test individual vertices to see whether they are inside or outside the clipping rectangle. Every vertex is assigned a binary code, called a region code, that identifies the location of the vertex relative to the boundaries of the clipping rectangle. Here is a description of the regions:



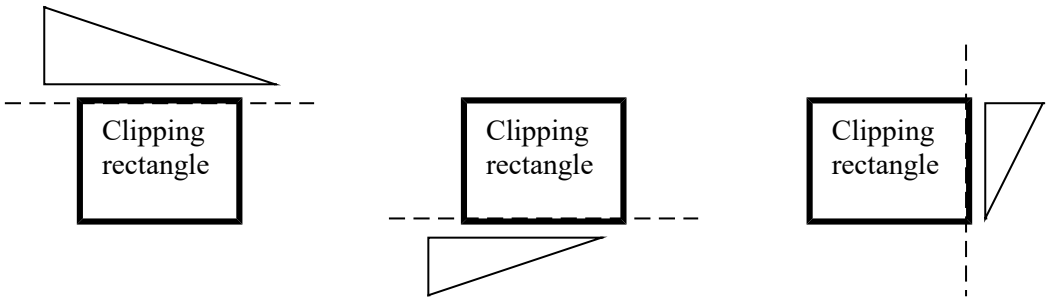
The region code can be related to the bit position as
bit 1: left
bit 2: right
bit 3: down
bit 4: up
bit 5: in front of screen (view plane).

A value of 1 in any bit position indicates that the vertex is in that relative position. Bit values in the region code are determined by comparing the vertices to the clip boundaries. Bit 1 is set if x < left boundary.

Once the region codes have been established it is possible to quickly determine which triangles are completely inside and which are outside.
The first stage is to perform a logical OR on the region codes of the three vertices,
Temp = (V0.RegionCode) OR (V1.RegionCode) OR (V2.RegionCode).
If the value of Temp= 00000, then all the points of the triangle are within the clipping rectangle, therefore the triangle will be displayed.
If the value is not 00000 then the triangle is definitely clipped, however part of the triangle may still be visible, therefore it cannot be discarded just yet. Here are some examples of situations where by all the vertices are outside the clipping rectangle, but some of the triangle is still visible. These will be passed to the next clipping process.



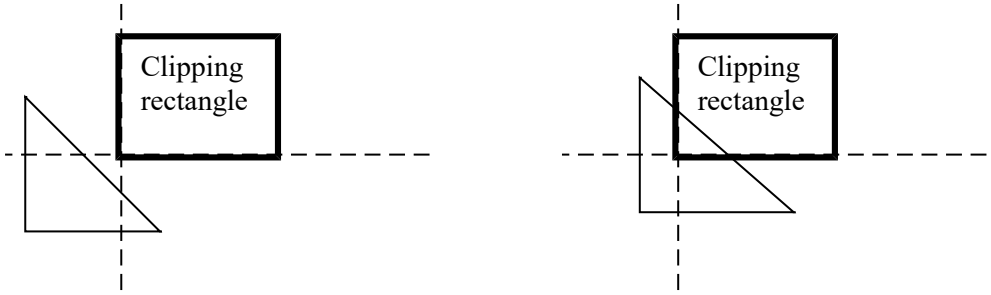
Only if all the vertices are in the same region can the triangle be classed as definitely outside the clipping rectangle. For example



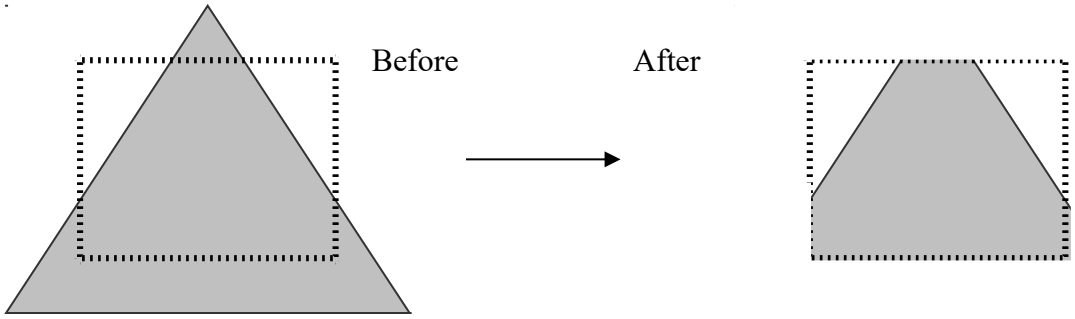
By performing a logical AND on the vertices region codes it is possible to determine whether they are contained within one region.
Temp = (V0.RegionCode) AND (V1.RegionCode) AND (V2.RegionCode)
If Temp is not equal to 00000 then the triangle is definitely out, and therefore will not be drawn. If Temp is equal to 00000 then part of the triangle may be visible, but this is not guarteed. The triangle will be passed to the triangle clipping process.

Triangle Clipping

If a triangle reaches this stage, it is still not known whether any part of it is visible or not. For example, here are two triangles that have been passed to the triangle clipping process. The vertices in both triangles have identical region codes, however one triangle is partly visible, while the other is not.

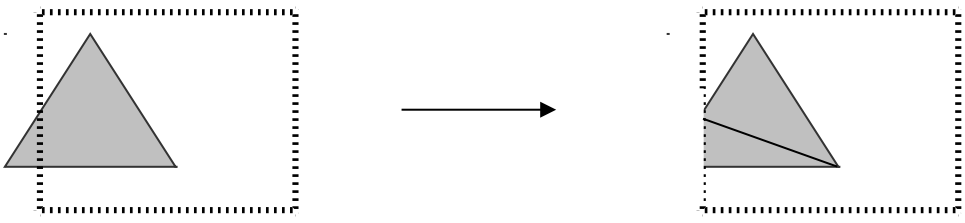


The aim of triangle clipping is to remove everything which is not in the clipping rectangle. For example:



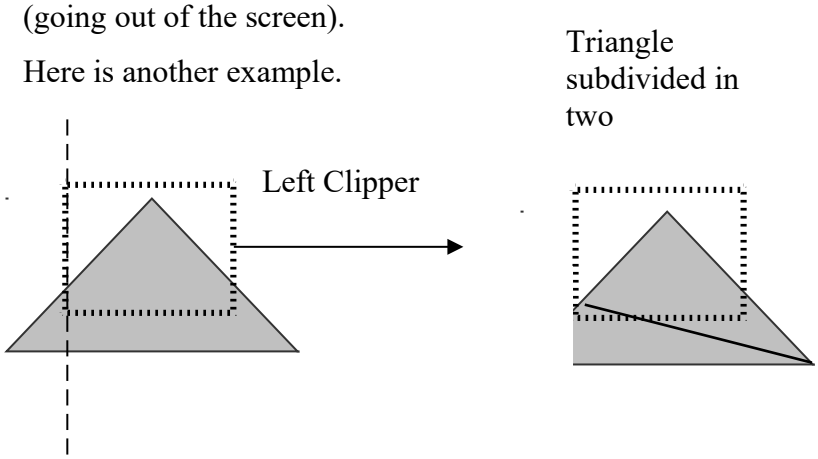
This problem is not particularly difficult to solve if the renderer could draw n-sided polygons. But as it can only draw triangles, the task becomes more involved.

The first stage involves creating five procedures, each clips a triangle against a specific boundary. When a triangle is clipped against just one boundary, the clipping routine may have to generate another triangle. For example, the triangle below is passed to the left clipping routing, and is broken down into two triangles.



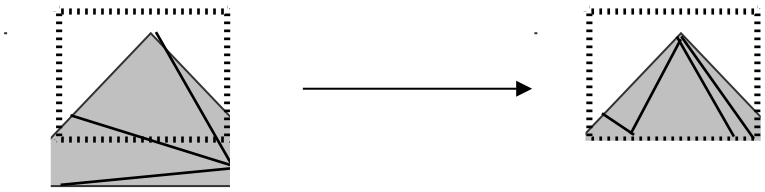
The order in which a triangle is clipped is, left, right, top, bottom and front plane (going out of the screen).

Here is another example.



Both triangles, are clipped by the right clipper. Generating another two triangles

All the triangles are then clipped against the bottom clipper. Producing the final image.



Starting with just one triangle the clipping routine has a final image containing five triangles. The clipping algorithm is similar to Sutherland-Hodgeman polygon clipping algorithm, except it is complicated by the fact it can only draw triangles.

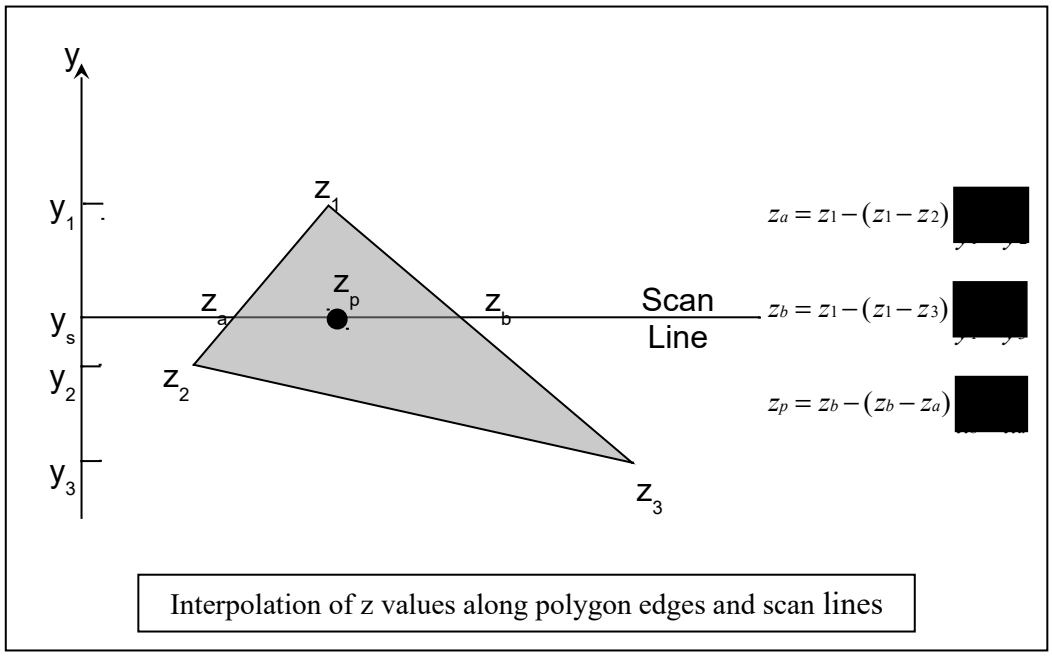
Depth Buffering

When drawing a 3D-scene, there has to be a way of determining which parts of the scene are visible from the chosen viewing position. The algorithms are referred to as visible-surface detection. The algorithm that most modern 3D-games use for visible-surface detection is depth buffering. Depth buffering ensures that objects in the distance do not appear in front of objects that are closer to the viewer. This method of visible surface determination requires no surface sorting.

The depth-buffer method involves using two buffers. A depth buffer is used to store the depth of pixels for each (x,y) position. A screen buffer is used to store the colours of pixels for each (x,y) position. Triangles are scan-converted into the screen buffer in arbitrary order. During the scan conversion process, if the depth of a pixel at (x,y) is closer than the depth value held in the depth-buffer, then the new pixel depth and colour replace the old values stored. In order to use the depth buffer, there first must be a way of calculating the depth value of each pixel.

To calculate the depth for each pixel on the surface of a triangle can be time consuming. However by using interpolation the process can be made a lot more efficient. The techniques described here are almost identical to the ones used to deign parts of Gouraud shading, Phong shading and texture mapping. They all use interpolation.

The depth buffer works by calculating the depth along each edge of the triangle. The depth value is then interpolated across between the two edges on each scan line. The diagram below describes exactly how to implemented depth buffering.



Interpolation of z values along polygon edges and scan lines

To interpolate between two points, for example Z1 and Z2 only requires one division with a number of additions. Here is an explanation:

From the diagram

Za = Z1 - (Z1 -Z2) * (y1 - yS) /(y1 - y2);

Or rearange slightly:

Za = Z1 -(y1 - ys) * (Z1 - Z2)/(y1 -y2);

(Z1 - Z2) / (y1 - y2) is constant therefore can be pre-calculated before scan conversion of that edge starts. The final equation for calcualting the depth between Z1 and Z2 is

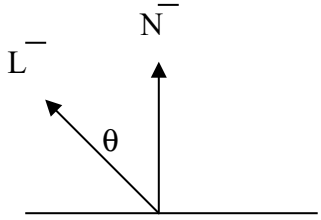
Za = Za -(y1-ys) * K

Where K is (Z1 - Z2) / (y1 - y2) which only needs calculating at the start of the interpolation.

Once the depth of each edge is calculated, the renderer then interpolates between the two edges across the scan line to produce the depth value of each pixel. The pixel can then be tested to see if it will be accepted into the depth buffer.

Flat Shading

This is the simplest method of shading and also the fastest. The intensity value for each triangle is calculated by using Lamertian reflection. The intensity is constant across the surface of the triangle. The reflected light intensity is proportional to the cosine of the angle between the normal of the triangle and the direction of the light.



For this approach to be valid the following assumption is true:

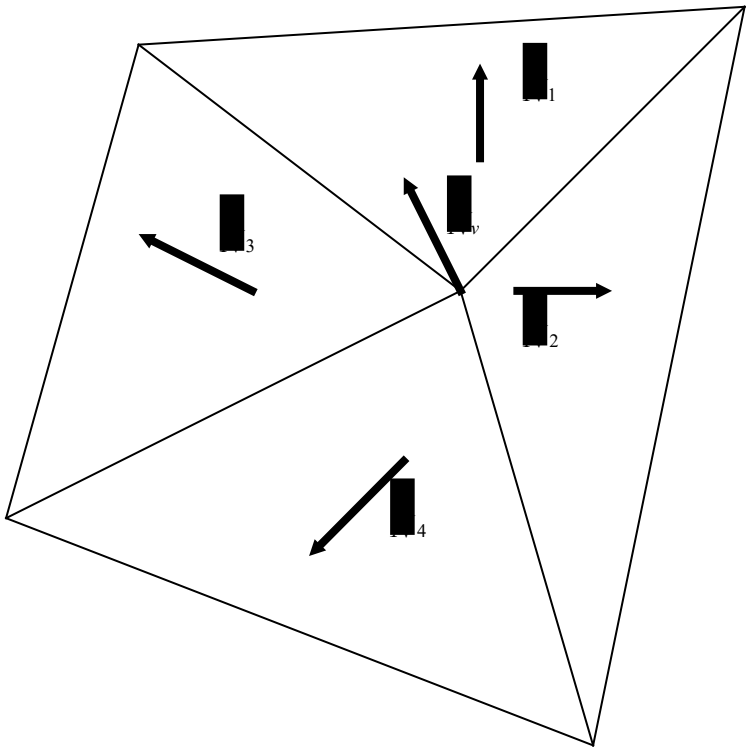
1. The light source and the viewer is at infinity, so that the angle between the normal and the light is constant across the triangle face.

Gouraud Shading

Gouraud shading renders a polygon surface by linearly interpolating intensity values across the surface. Intensity values for each polygon are matched with the values of adjacent polygons along common edges, this eliminates the intensity discontinuities of flat shading.

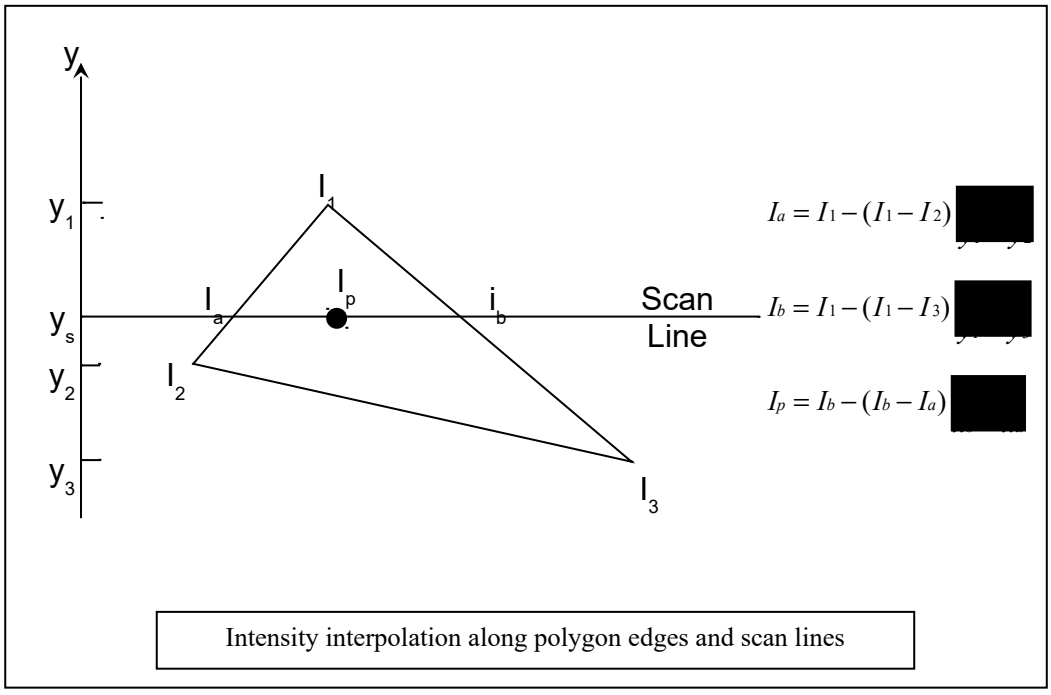
Each surface is rendered with Gouraud shading by performing the following calculations:
Determine the average unit normal vector at each polygon vertex
Apply an illumination model to each vertex to calculate vertex intensity
Linearly interpolate the vertex intensities over the surface of a polygon

At each triangle vertex, the normal vector is calculated by averaging the surface normal of all triangles sharing that vertex. For example:



Normalised polygon surface normals

After calculating the intensities of each vertex, the intensities are interpolated across the triangle. The equations used to perform the interpolation are very similar to the ones used to perform depth buffering. Thus making the implementation easier. Here is a diagram containing the equations used to interpolate the intensities across a triangle.



Intensity interpolation along polygon edges and scan lines

Although Gouraud shading removes the intensity discontinuities associated with the flat shading model, it does have its disadvantages. Highlights on the surface are sometimes displayed with anomalous shapes, and the linear interpolation can cause bright or dark intensity streaks , called Mach bands. These problems can be overcome by using Phong shading.

An example of flat and Gouraud shading can be found in Appendix B. These two images were rendered with the 'Velcocity 3D' engine in its high-resolution mode. The object used is an oil filled lamp.

Phong Shading

Phong shading displays more realistic highlights on a surface and greatly reduces the Mach-band effect. A polygon surface is rendered with Phong shading using the following steps:

- Determine the average unit normal vector at each polygon vertex
- Linearly interpolate the vertex normals over the surface of the polygon.
- Apply an illumination model to each point to calculate the pixel intensity

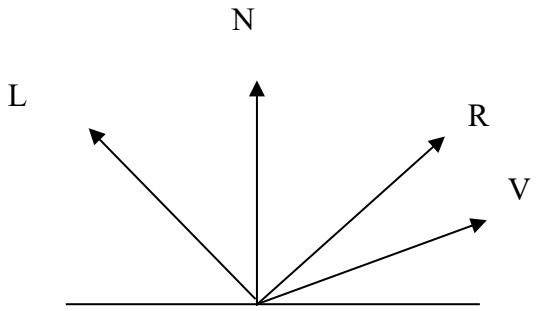
The first stage of Phong shading is already implemented as part of the Gouraud shading.

To linearly interpolate the vertex normals over a triangle similar equations to Gouraud shading should be used. Phong shading is a lot slower than Gouraud because instead of interpolating just one value (the intensity), Phong interpolates the three values of a normal.

Lighting Models

Lighting models are used to calculate the intensity of light seen at a given point on the surface of an object. Currently flat shading and Gouraud shading are designed to use diffuse reflection. Diffuse reflection is fine for surfaces that are dull, such as chalk, but it is inadequate for shiny surfaces such as metals. When looking at an illuminated shiny surface, such as polished metal, a high-light or bright spot, can be seen at certain viewing directions. This phenomenon is called specular reflection. It is a result of total, or near total reflection of incident light. A model for calculating specular reflection is called the Phong specular-reflection model.

The intensity of specular reflection depends on the material properties of the surface and the angle of incidence.
Here is a diagram of specular-reflection occurring on a surface.



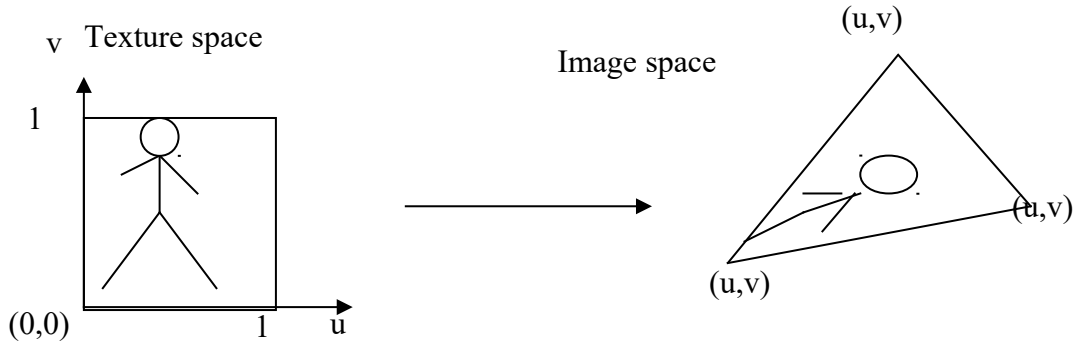
L is the light vector, N is the normal to the surface, R is the reflection vector and V is the view vector. The formula for specular reflection at a point is:

I spec = Ks (V. R) ^n

Ks and n are dependent on the material. Therefore to perform specular-reflection the value of V.R needs to be calculated. The value of V is already known, the value of R can be found using this equation:
R= (2N . L)N -L

Texture Mapping

Texture mapping is a way of applying a raster image to a polygon to give a more realistic appearance. The design to perform texture mapping uses linear interpolation. This time to interpolate values of u and v across a triangle.



The values of u and v are calculated within 3D Studio and assigned to each vertex of the triangle. Once the values of u and v are interpolated across a triangle the only operation left to perform is to look up the colour of the texture at that particular point. A pixel within a texture is called a texel (textured pixel).
As an addition to the texture mapping a method called bi-linear interpolation has been included into the design.

Bi-linear interpolation

When texture mapping a polygon sometimes the situation occurs where individual texels stretch over several pixels. This starts to happen as soon as the image space, becomes larger than the texture space. One way to get around this problem is to use a larger texture, however this is not always possible.

Bi-linear interpolation attempts to solve this problem by averaging the colour values of the pixel being plotted over the four closet texels. The overall result is that a triangle with a texture map on appears less blocky and more realistic.

Environment Mapping

Environment mapping is becoming popular in 3D driving simulators to make the cars look more realistic. In the context of a driving simulator it is used to show the surrounding environment such as the sky and trees reflect from the side of the car. The design for performing environment mapping is fairly simple. It involves loading a texture, such as a landscape, calculating u and v texture values for each vertex, then texture mapping the texture on to the object.

When the normal for a vertex has been calculated, the values of its texture co-ordinates u and v are also calculated as follows,

```
U= (1/2)+Normal.x*(1/2)
V= (1/2)+Normal.y*(1/2)
```

This is not the most accurate way of performing environment mapping, because the environment is not actually being mapped on to the car, just a texture map of a landscape.

Super Sampling

To remove the jagged appearance of the edges of the triangles it is possible to use a technique called super sampling. Super-sampling makes use of the property that alias effects decrease with resolution. The 3D engine generates a picture at a higher resolution than the output device can handle. The image is then mapped on to the real screen by averaging the intensities over a certain area.

Super sampling is slow, the 3D-engine should only use it when operating in its high resolution mode to produce high quality test images.

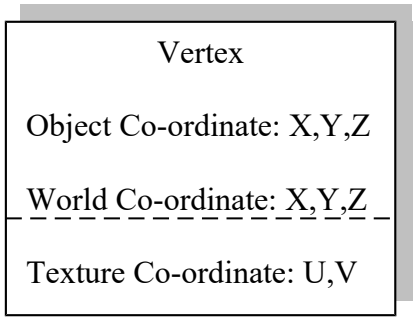
When the 3D-engine is in high-resolution mode all the frames are rendered at a resolution of 2400 * 1800. The frame is then mapped down to a resolution of 800*600. In 3D graphics memory limitations can become a problem with super-sampling. The problem arises because not only do you require extra memory to hold a high-resolution image, but the depth buffer also needs to be set at the same high resolution. For example, in the high-resolution mode the depth buffer is around 17 MBs.

Parser and data type changes

The design of the original parser needs modifying slightly at this stage to accommodate for objects with texture co-ordinates. A texture mapped object has two extra parameters at the end of each line, the u and v co-ordinates.

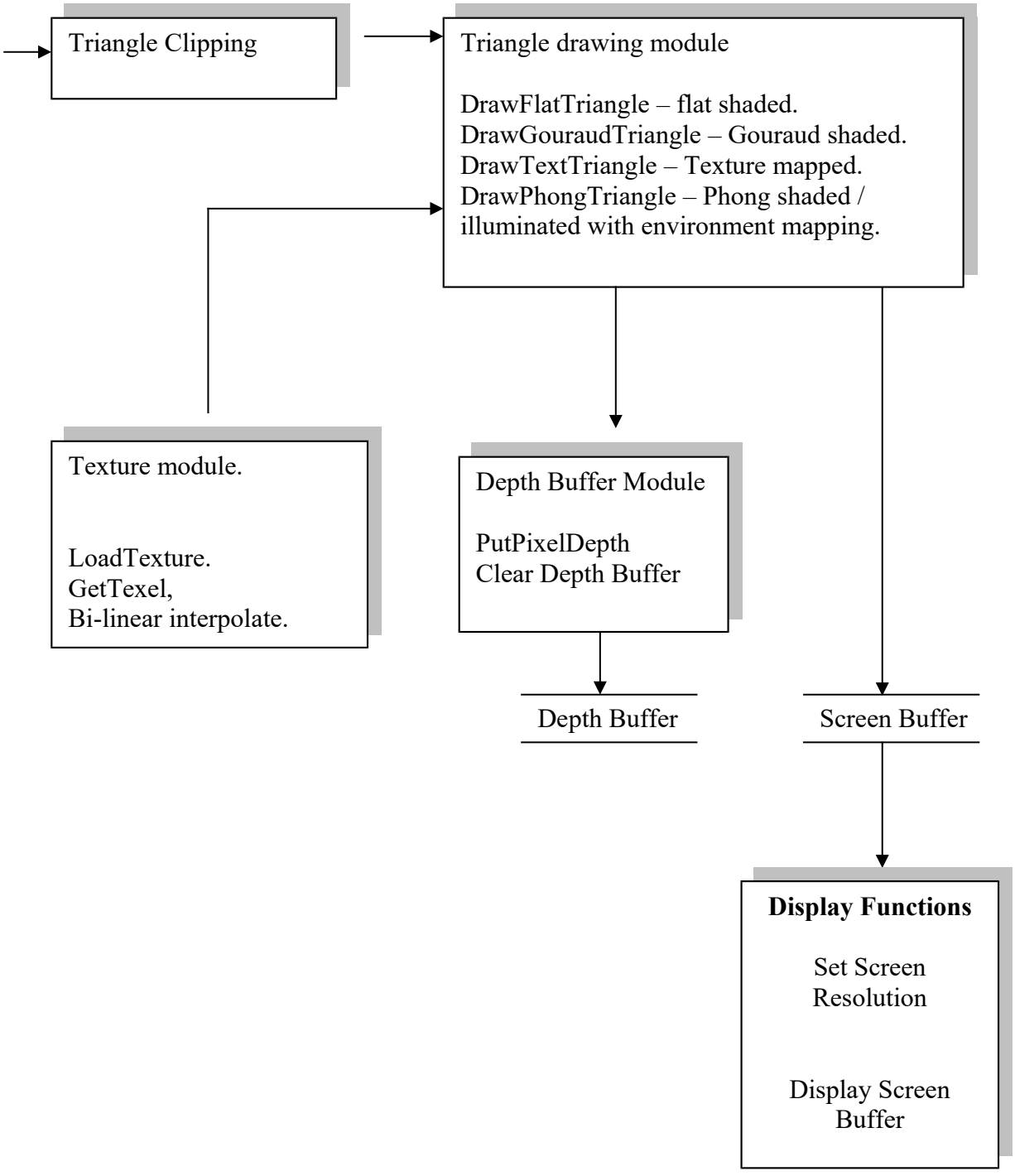
```
Vertex 240:  X:14.010956     Y:-250.569992     Z:102.078682
U:0.733949     V:-0.204104
```

To accomadate the changes the data type of a vertex also needs changing slighlty so that it now includes the texture co-ordinate. For example:



Design Overview of the 3D-Renderer

To help understand how the various parts of the renderer fit together, here is an overview:



Design of Display Functions

The requirements for the display functions are:

- Capable of setting various screen resolutions, ranging from 320x200 to 800x600
- Capable of copying the screen buffer into the graphics card and displaying it on screen.

Most compilers provide functions for setting the screen resolution. For the prototype version of the 3D-engine, it uses a Modula 2 command called, SetVideoMode. For the Visual C++ version it uses a command called `SetDisplayMode`.

When copying the contents of the screen buffer to the graphics card it is essential that the process is done as quickly as possible. At a screen resolution of 800x600 there are 480,000 pixels. If the process is slow, it will take a long time to copy the entire screen buffer.

The best way to perform such a task is to use a straight memory copy, from main memory into the memory on the graphics card. Avoiding using any compiler provided functions such as put pixel. For the prototype version, just one command built into Modula 2 called FastMove is used to copy the entire contents of the screen-buffer in to the graphics card. Here is the pseudo code for it.

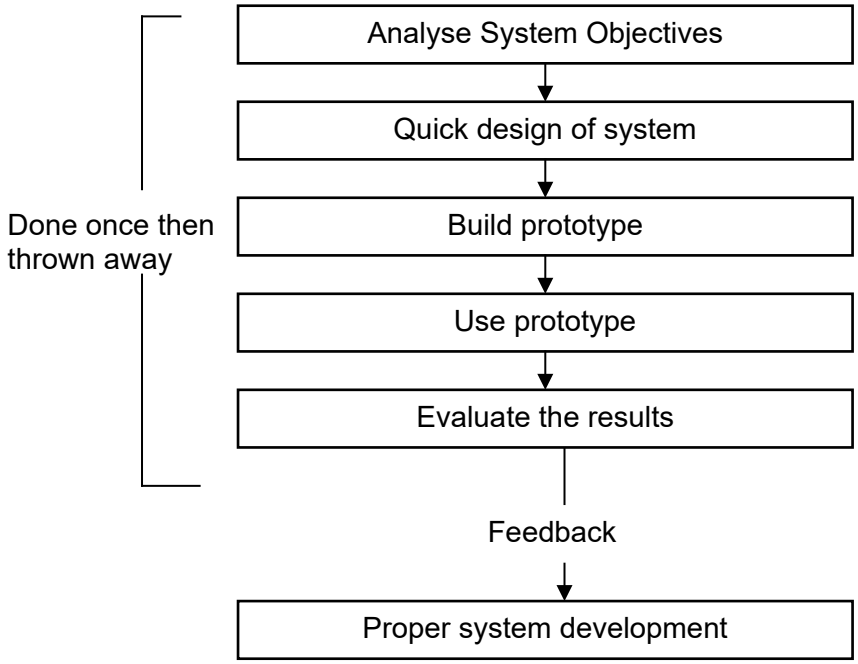
```
FastMove(ADDRESS(ScreenBuffer),ADDRESS(GraphicsCard^),64000);
```

The above line moves 64K of data used to store the screen at a resolution of 320x200 with 8 bit colour. To speed up the final C++ 3D-engine, the function is written in assembler, to ensure it is as fast as possible.

As soon as the memory has been copied into the graphics card, it will be visible on screen. To avoid screen tearing, the memory copy command should be used straight after the monitor has just completed a full scan of the screen. The graphics card provides a register that holds the status of the monitor. The program has to continually poll it until the register is set to a value that represents the monitor has just completed a scan.

Development and Implementation

To develop the 3D engine, the throw-away prototype model has been used.



The prototype version of the 3D-engine is written in TopSpeed Modula 2. Unfortunately not all the functions found in the final version are present in the prototype. This was due to memory limitations. Modula 2, compiles 16 bit code for DOS. This means that any variable created within the 3D engine can not be greater than 64K. This gives a limit to how many polygons a single object can contain. The other memory limit is caused by the fact the program when executing can not be larger than the 640k DOS limit. For these reasons the prototype can not do the following: Depth Buffering, texture/environment mapping and any shading other than flat shading. It renders at a screen resolution of 320 x 200, with a 256 colour look up table.

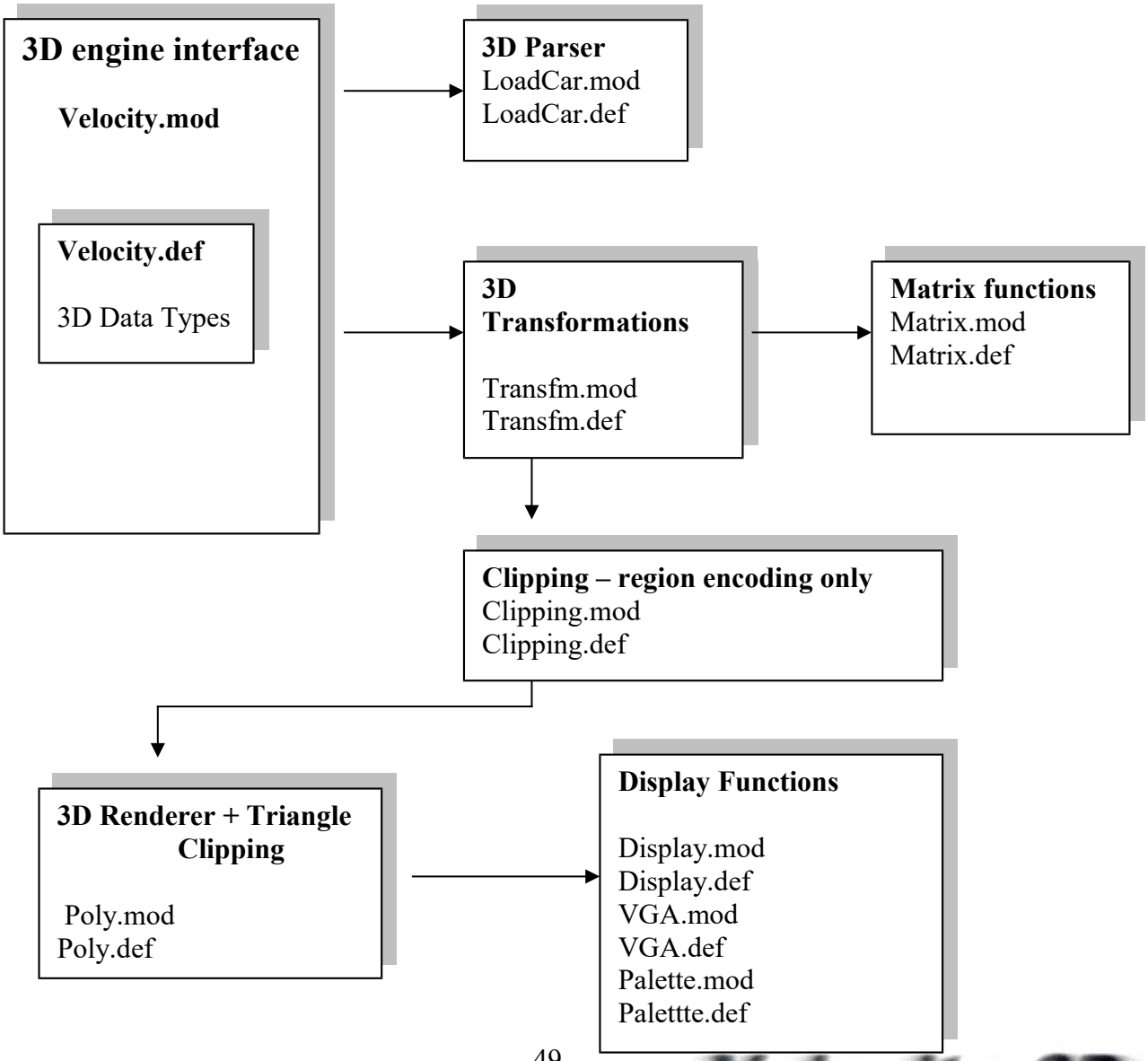
The final system has been written in Microsoft Visual C++ (Version 4) with some inline assembler. The system runs as a Windows application, with a stub module designed to test it. Originally the final system was going to be written entirely in Assembler, but due to the complexity of 3D-graphics and the time constraints this was unachievable.

Difficulties

The most difficulty experienced in the project involved the triangle clipping module design and implementation. There is a wide range of material available about clipping n-sided polygons, however, there seems to be an apparent lack of material on performing 3D-triangle clipping. A possible reason is that most modern 3D-games make use of hardware chips on a graphics card to clip and draw triangles. Therefore removing the need to perform triangle clipping in software.

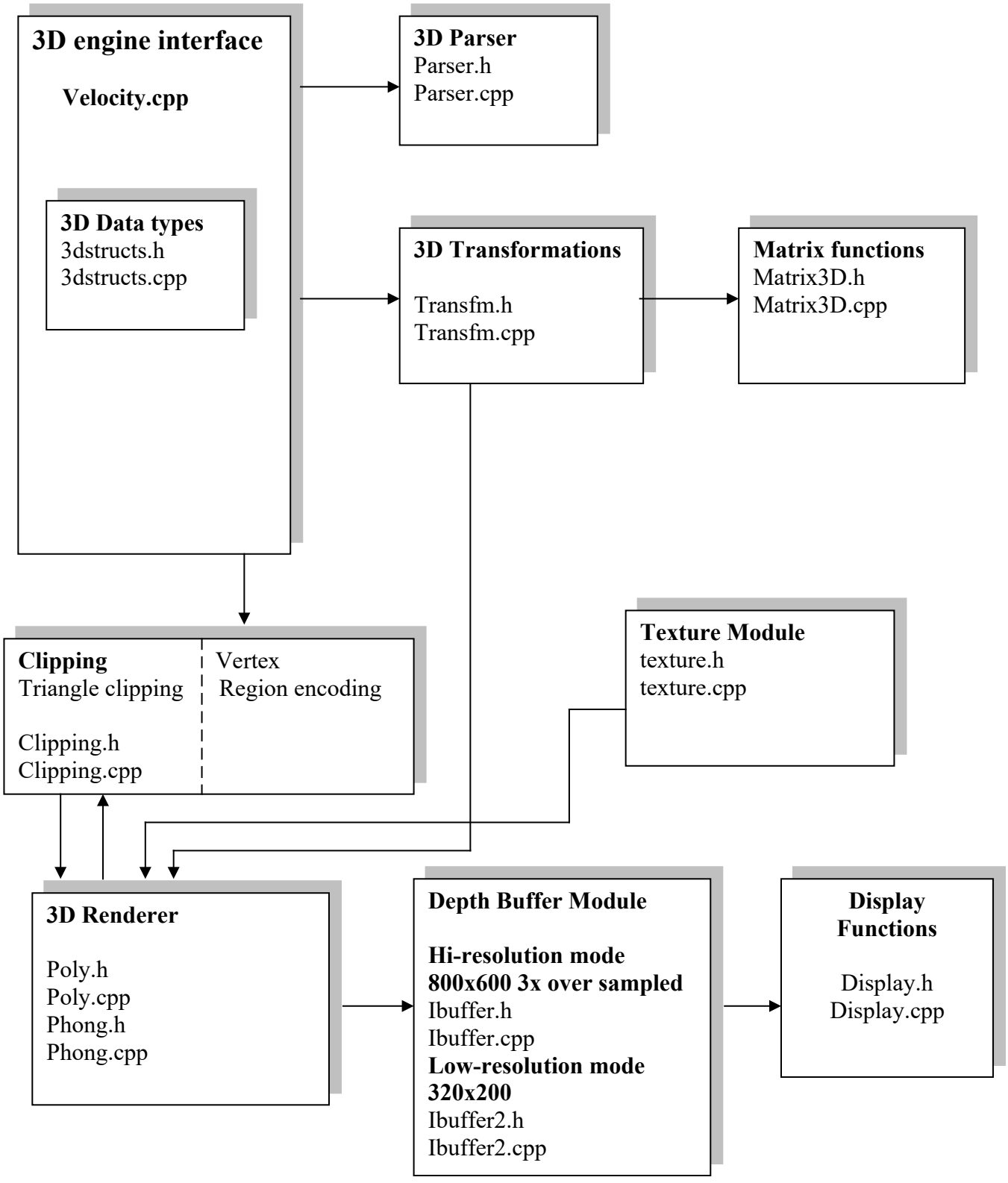
Prototype component design.

This diagram shows the various components of the 3D-engine, and how they relate to the prototype files.



Final system component design.

This shows all the components of the final 3D-engine put together. All the files are listed here. Changes from prototype: Triangle clipping is now in a separate module to the renderer, texture and depth buffer module implemented.



Program Specifications

Specification of the prototype
Operating enviroment: Dos / Windows 95/98
Computer required to run a demo: Intel 80486, 4Mb RAM, VGA graphics card.
Compiler: TopSpeed Modula 2.
Output. Output to the screen, set at a resolution of 320x200 with 8 bit colour.
Loads: .ASC 3D studio files

Specification of final system
Operating enviroment: Windows 95 /98
Computer required to run : Intel Pentium, 32Mb RAM, 4MB SVGA graphics card.
Compiler: Microsoft Visual C++ Version 4.
Output. Output to the screen, set at a resolution of 320x200 or 800x600 24 bit colour.
Loads: .ASC 3D studio files

The final system operates within the Windows 95/98 operating system. In order to use the graphics card in Windows, the program uses a Windows API called DirectDraw. DirectDraw allows the 3D-engine to interface with the graphics card, indepent of the make or model.

In the main 3D-engine interface file “velocity.cpp” there is a certain amount of Windows intialisation code. This project is not concerned with anything other than the 3D-engine. It is designed to be embedded within a 3D-driving simulator. Therefore there will be no explanation on how the code to create a window works.

Testing and Results

As each component of the 3D-engine has been implemented, extensive testing has been carried out. Four different testing strategies have been used in total.

The first strategy is unit testing. It involved writing drivers to test individual modules and functions. The drivers are designed to test modules to make sure the interface is working, the local data structures work and boundry conditions are correct. It also attempts to test all independent paths. In a 3D-engine it is essential that each module works correctly, if just one module gets an error then it may cause the whole engine to fail.

The second strategy used is called bottom-up integration. It involves the following steps:

1. Low level modules are combined into *clusters* that perform a specific function.
2. A driver is written to co-ordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

The third strategy used is validation testing. Validation testing provides final assurance that the 3D-engine meets all functional, behavioural and performance requirements.

The final strategy used is system testing. System testing verifies that all elements mesh properly and that overall system/performance is achieved..

Results of Validation testing

The final system is capable of meeting all the following requirements defined in the analysis.

- Capable of reading files from a disk containing 3D geometric representations of objects, then loading them into memory.
- Capable of performing 3D transformations on 3D objects.
- Capable of rendering the 3D objects, to be displayed on a monitor.
- Capable of interfacing with the graphics card to set various screen resolutions.
- Capable of producing smooth animation, without any screen tears.
- Capable of executing on P166 PC running Windows 95/98 with 32Mb of RAM.
- When running the 3D-engine on a 300Mhz Pentium II it must be able to draw at least 50,000 texture mapped triangles to the screen every second. This means around 1600 triangles per frame, when running at 30 frames per second.

The last requirment was tested by rotating two texture mapped spheres both made from 528 faces around the screen at 60 frames per second. This equals 63,360 textured mapped triangles drawn every second, exceeding the required amount. This test was carried out in the low resolution mode.

Errors found

One of the best ways to test the 3D-engine is to compare what an object looks like within 3D Studio, then compare it to how it looks after being drawn by the 3D-engine. It was found during testing that although, the objects were displayed correctly, the orientation was not correct. The problem was rectified by swapping the y and z values around of each co-ordinate as the parser read an object in from a file. It was assumed that the must have been a problem in the transformations stage causeing the orientation to go wrong. However, later on during development the real reason for the problem was discovered in a book on how to design 3D graphics, it contaians a the following sentence,

“Tip: 3DS Studio swaps the Y and Z values of all vertices when it writes a .ASC and .DXF files by defualt.” [reference]

Therefore around a weeks debugging in total was waisted trying to solve a problem that didn’t exist.

System testing and Results

To test the functionality of the 3D-engine several real-time demos have been created and executed. The aim of the testing was to try to make sure every single stage of the 3D-engine was working correctly.

For the purpose of this report a number of screen shots have been taken of objects drawn by the 3D-engine in high-resolution mode, using super sampling. They are included in Appendix B, titled ‘pictures’.

The screen shots show the use of the various shading algorithms described in the design, including environment mapping They are proof that every module is working correctly, as specified. If just one module was working incorrectly it would be almost impossible to display an image to the screen. To show the parser is working correctly three different objects are present in the results, an oil filled lamp, a Porsche 911 and finally a 55 Porsche.

The first slide features two oil filled lamps, one is flat shaded, while the other is Gouraud shaded. The slide is designed to show difference between the two shading algorithms. The second slide features two oil filled lamps, one is phong shaded and uses the specular reflection model to show a high-light. The second lamp is environment mapped. The texture map used to environment map the lamp is just an image of a sunset.

The third slide features a two lamps, both of which use a combination of phong shading, and environment mapping. This time one of the lamps uses an environment map of a picture taken of the sky.

The remaining slides contain pictures of a Porsche 911 and a 55 Porsche. The shell of both cars are environment mapped to give a more realistic appearance. The windows of the Porsche 911 use specular reflection to produce a glass like appearance. The tires of both cars are phong shaded, as are do the headlights of the Porsche 911.

7.Summary

Costings/Estimates

Cost estimation using 3-Point LOC estimation technique

Main Requirements for a 3D graphics engine written in Modula 2

Lines of code	Optimistic	Most likely	Pessimistic	EV	Actual
3D data parser	50	100	200	108.3	230
Object manipulation	200	400	500	383.3	540
Polygon scan conversion	50	100	200	108.3	820
Texture mip-mapping	200	400	500	383.3	0
Depth buffering	200	300	500	316.6	0
					+more
			Total	1300	3700

EV = optimistic + most likely * 4 + pessimistic

Main Requirements for final 3D graphics engine written in assembler.

Lines of code	Optimistic	Most likely	Pessimistic	EV	Actual (Visual C++)
3D data parser	100	200	300	200	767
Object manipulation	400	500	600	500	500
Polygon scan conversion	100	150	200	150	2000
Texture mip-mapping	300	500	600	483.3	200 (normal mapping)
Depth buffering	200	400	600	400	200
					+more
			Total	1733	8500

Historic data indicates 620 lines of code can be written per month.

Estimated time for prototype = 2 months ,1300 LOC

Actual time for prototype = 6 months, 3700 LOC (almost 620 month)

Final system LOC = 8500 in 2 months, (LOC 4250 month)

Total lines of code estimated = 1300 + 1733 = 3033

Actual lines of code = 3700 + 8500 = 12200

Total time estimated = 3033 /620 = 4.89 months.

Actual time required if programming 620 LOC a month = 12200 / 620 = 19 months.

Costing Summary

The estimations made in the Preliminary were greatly underestimated. This is because it is hard to estimate the work required to produce a project if the subject area is new to the programmer. For this project the complexity of 3D-graphics were underestimated, resulting in low estimates for the overall lines of code for the project. The total lines of code is four times larger than estimated. One of the reasons for this is that more features have been added to the 3D-engine than first concieved.

Conclusion

The final 3D-engine is fully working The requirements for it in terms of both functional and non-functional have all been met. The only aim which has not been achieved is to program the 3D-engine in Assembler. This decision, given the time constraints was over ambitious. However, it is still believed that in order to achieve the maximum performance, Assembler should still be used.

In terms of performance, the 3D-engine is fast when running in a low resolution mode, achieving around 60 frames per second with an average 3D scene (2000 polygons) on a Pentium II 333. Despite this, it has trouble maintaining a frame rate of above 15 frames per second when performing Phong shading on just small objects. Therefore if the 3D-engine is to be integrated in to a driving simulator, the use of Phong shading will have to be limited.

The objects rendered in the high-resolution mode look fairly realistic. By using super-sampling the jagged edges normally associated with graphics have been smoothed out. The only problem with super-sampling is that it is slow, and requires a lot of memory. Therefore its use in a driving simulator may also be limited.

If the 3D-engine were to be continued, in view of recent technology the aim would be to add hardware rendering to it, as an option. This would allow a graphics card with hardware acceleration to perform most of the triangle rendering itself. Due to the modular design of the 3D-engine, it would just require a module adding, to be interfaced with the transformations module. Most 3D graphics cards include hardware depth buffering as well, this would mean the depth buffer module written may not be required.

Another feature which would be added is object hierarchy. Currently objects are modelled as individual entities, there is no connection between them. In a driving simulator the wheels should be separate objects that are connected to the car via some form of link. When the car is moved forward, so should the wheels. The wheels should then be able to rotate independently of the car, to simulate turning. Currently this is not supported.

Summary

As already stated the 3D-engine meets all the requirements laid down. Despite this, there are a number of future expansions that can be carried out to improve the project. The most essential one is the use of hardware 3D accelerated graphics cards. The requirements for the 3D-engine did not include the ability to make use of 3D graphics cards because when the project was started a year ago they were not widely excepted. Now, however almost every OEM PC made contains a graphics card which can perform 3D hardware acceleration.

References

[1] Playstation, produced by Sony.

[2] Nintendo 64, produced by Nintendo.

[3] AMD K6 3D NOW, produced by AMD.

[4] Screamer 2 is by Milestone. Published by Virgin Interactive Entertainment (Europe) Ltd. 2 Kensignton Square, London, W8 5RB.

[5] Motorhead is published by Gremlin Interactive LTD.

[6] Grand Prix Legends, published by Sierra.

[7] Colin McRae Rally is written and published by Codemasters.

[8] Daytona 2, is produced by SEGA. Polygon figures for Daytona 2 came from June 98 edition of EDGE magazine. Published by Future Publishing.

[9] Windows 95, is made Microsoft.

[10] 3D Studio Release 4c1 by Autodesk.

Bibliography

D. Hearn, M. Baker (1997) Computer Graphics, Prentice Hall.

J. White (1996) Designing 3D Graphics, Wiley. P17

J. Goes (1996) 3D Game Programming, Corilolis Group.

P.Burger, D. Gillies (1994) Interactive Computer Graphics, Addison Wesley.

B. Cornelius (1994) TopSpeed Modula-2, Addison Wesley.