CALLBACKMANAGER

Our callback manager class is made up entirely of static function allowing interaction between the glut callbacks and the Game class. We didn’t want to mix the Game class with static and non static classes, so we made a separate CallbackManager class which stores a pointer to the ‘active’ game instance (the one who should receive the callbacks) and would forward the callbacks to it.

GAME

The game class is responsible for handling the game, while not containing any game specific code, enabling it to be reusable for other glut projects. The game class is responsible for forwarding the callback functions to the GameState class, which is an abstract class that handles a ‘state’ of the game. The game class also caches some of the information from the callbacks, such as the mouse co-ordinates, whether a key is down or not, and whether the mouse left and/or right mouse buttons are down.

GAMESTATE

In our project, we used three states – one for Shays world, one for the Minicar world, and one for the credit screen. The GameState class contains a pointer to the game so it can easily obtain cached information from glut (see above). The class contains a constructor and a destructor to allocate and free memory for textures, models, and other ‘should only be loaded once’ items, as well as an ‘onEnter’ and ‘onLeave’ callback for when the game class switches scene to set render states and set initial parameter for variables.

DETAILS OF THE PLANS AND MEASUREMENTS USED

David and Troy took photos and measurements for his tavern area. They drew sketches (seen in the directory /root/daves sketches/).

For the part John modelled (from the ECL side to the food court steps) he used David’s photos as references, and modelled mainly by counting bricks and working out ratios of objects from photos.

WHAT ISSUES DID WE ENCOUNTER?

We wanted to have real-time reloading of CUO models in our world to make the modelling far easier. It meant we would not have to recompile to see changes. We implemented this quite easily because of the existing Mesh Manager class, we could simply update each mesh’s mesh using the Mesh manager when pressing a key.

We decided to use the state pattern to separate game states, however this conflicted with glut’s call back functions which could not accept a member function pointer, only an ordinary function pointer. To fix this we created a Callback Manager class which is full of static functions which can be accepted by glut, and these functions simply call our current state’s real function.
Originally we were determining frame time by using the clock(). However this produced jittery results which John noticed and diagnosed quickly as the time difference being 0. We are still unsure as to why this happens but it also happened in John’s ICT215 game. To solve this problem we simply changed the time related functions to glutGet(GLUT_ELAPSED_TIME).

When we started loading .png files with transparency strange things started to happen and the transparent part of the image was either completely black or white. David quickly identified it as a problem with alpha textures and mipmaps, and implemented pre-multiplied alpha channels which fixed the problem.

The terrain class was originally using GL_COLOR_MATERIAL for all attributes, which meant that the colour of the terrain was determined by the colour map alone. However this caused bad results with the lighting for the rest of the world, and had to be changed so that the gl colour material only affected the diffuse colour, not the ambient colour of the terrain using this function call:

```c
glColorMaterial( GL_FRONT_AND_BACK, GL_DIFFUSE );
```

Note: specific details about problems with shays code can be found near the end of this document.

**SPECIAL FEATURES:**

**SPHERE / 2D WALL COLLISION DETECTION**

In our game world we used 2d walls for collision detection. A 2d wall in our world is a set of 2 points, and our player is represented by a bounding sphere. The sphere-wall collision algorithm involves first checking which the player is closest to, the wall itself, it’s start point or it’s end point. If the dot product between the vector from the middle of the sphere to the first point and the wall’s normal vector is less than 0 then the ball is closes to the start point. If the dot product of the vector from the balls centre to the second point and the wall’s normal is less than 0 then the sphere is closest to the second point. Otherwise it is closes to the wall itself.

If the sphere is closest to the end points, it’s an easy sphere-point test to find the amount the sphere must move back.

If the ball is closest to the wall itself then the amount to move back can be found by checking the distance of the spheres centre to the plane defined by the two wall points. This is the dot product between the walls normal and the vector between the start point of the wall and the spheres centre. If the distance is less than the spheres radius, it is colliding and must be moved back along the walls normal.

Classes with this: Wall, PhysWall, BoundingSphere

**CUO MODEL FORMAT**

The CUO model format is one we invented which is a much more ‘human readable’ format than the obj format. Texture co-ordinates are specified with a t followed by 2 numbers separated by whitespace, vertices are specified with a v followed by 3 numbers separated by whitespace, polygons specified with a p followed an index of each vertex used, and polygon texture co-ordinates specified with a pt followed by an index of each texture co-ordinate used. The ‘end’ keyword specifies that a new section of the file has begun, and the vertex
and texture co-ordinate index's used in the pt and p commands refer to vertices and texture co-ordinates defined after the end keyword.

## STATE PATTERN FOR GAME STATES

We implemented the state pattern in our game to handle different game states (levels). This allowed us to abstract away all the common properties related the a game level, world and state; Such things as glut call backs. Using this state pattern made it extremely easy to add new states and to change states on certain events. However, because glut call back functions cannot take member functions as parameters we had to create another class (CallbackManager) which is full of static functions that would then call the member functions on the current state.

## TERRAIN CLASS

Terrains are a special 3d modelling technique where a landscape is loaded entirely by a 2 dimensional image known as a ‘height map’ which contains a spectrum of colours, black representing the lowest and white representing the highest. The terrain is rendered as a segmented plane where the y position of each vertex is determined by the value in the grid. Our terrain class encapsulates this technique and adds some additional functionality, such as a colour map which loads an additional image the same size as the height map which specifies the vertex colour of each point. The terrain class can also load a texture and specify how many times the texture should repeat itself along the x and z axis. We use our class in the Minicar world to render the sand.

## PATH BUILDER

We created a path builder class to help with the creation of wall points for our Minicar world. The path builder takes a set of points that form a path, and from these, it generates smooth wall points on either side of the original points. The class was designed in a very reusable way, as it does not have anything to do with the Wall class, it simply takes a set of points and returns another set, and can be used in any other general place where you would need such functionality.

Below is a diagram of what the path builder generates, the red path is the original path, with the points as bold circles. The blue path shows the generated points. Notice how they are smoothed:
**SKY BOX**

The skybox is a technique used to create the illusion of a full spherical texture by using a 6 sided cube known as the sky box. It renders with such a transformation that it is only rotated but never translated, giving it the appearance of infinite distance. Our Skybox class creates a skybox instance and can hold 6 different textures, one for each side of the cube. As long as it is rendered first will be able to correctly orientate itself no matter what the model matrix contains or what render states are being used. It will also restore all render states so as not to affect the body of code it is being used in.

**ALPHA BLENDING**

In order to properly render transparent and semitransparent textures, we incorporated alpha blending which enables a texture to have an alpha channel and let the background show through the polygon. We used a special method known as ‘pre-multiplied alpha channels’ where the red, green and blue channels are pre-multiplied by the alpha channel before being loaded into the graphics card. The formula for rendering blended pixels is

\[ \text{pixel} = \text{dest} \times (1 - \text{src}_\text{alpha}) + \text{src} \times \text{src}_\text{alpha} \]

By multiplying the source by the alpha value, we remove the \( \text{src} \times \text{src}_\text{alpha} \) part of the calculation. Doing this makes the blending associative and therefore gives linear interpolation much better results. The final equation is:

\[ \text{pixel} = \text{dest} \times (1 - \text{src}_\text{alpha}) + \text{src} \times \text{one} \]

The results were incredibly noticeable with our mip-maps. Before we used pre-multiplied alpha channels, the fence on the Tavern began to develop a white glow around it as it got smaller eventually causing the fence to be a white glow in the distance. After using pre-multiplied alpha channels, the fence looked perfect and remained the same colour as you got further away from it.

**ANISOTROPY**

‘….is a method of enhancing the image quality of textures on surfaces that are far away and steeply angled with respect to the point of view. Older techniques, such as bilinear and trilinear filtering don’t take account of the angle a surface is viewed from, which can result in aliasing or blurring of textures. By reducing detail in one direction more than another, these effects can be reduced.’

We incorporated the glee library to our project to get access to the extension functions in OpenGL, and applied anisotropy to our textures and shays textures. The results were simply amazing! The windows used to become really blurry when you were looking at them at an angle, but anisotropy fixes this and causes the mirrors to render really sharp.

**VEHICLE PHYSICS:**

Rather than using a realistic physics system like most racing games, we instead went for a cartoony physics system that was highly inspired by the game Diddy Kong racing. Our objective was to create ‘fun vehicle physics’. The physics can be explained with the aid of a diagram:
Basically the vehicle has a forward/backwards component, and a left/right component. When you accelerate forward, the forward/backwards component has acceleration added to it. When you are not driving forward, deceleration is subtracted from it until it reaches 0.

These values act upon the position of the vehicle based on their orientation:

The left/right component behaves in the same way as the forward/backwards component, and are used to create the illusion of drift. When you turn to the right at the maximum turn angle, the left component of drift is increased. When you turn right, it’s vise versa.

Finally, the angle that these components are based on is the angle of motion. Angle of motion increases and decreases as you turn right and left. The angle the vehicle faces is the angle of motion plus the turnspeed. This means if you are turning right, the car will slightly overturn, and ask you stop turning right, the car rotates back towards the angle of motion. This has no effect on the driving except making it more visually pleasing and giving you a strong sense of turning right and sliding.

All these elements combined together create what we believe to be a fun and enjoyable physics system.

The artificial drift system allows the player to gain an advantage if they drift as their car is moved ‘forward’ by the speed and sideways by the drift, and if the vehicle is facing in such a direction that they move in the same direction as the track then they will actually be moving faster than just driving forward. However, drifting is a lot harder to control, and if the player collides with a wall they lose a lot of speed due to friction.

**RIBBON**

The ribbon class allows you to create a 3d ribbon with a collection of point pairs. In our game, we used the ribbon feature to create the tire tracks on the ground. First we found the two points by taking the location of the tire, working out where it was facing, and getting two points in world space that are from the left and right of the tire. We then set the y components of both these points to be just above the terrain. The ribbon class manages the points and renders a quad strip with the opengl commands. It also uses a circular buffer which means the data the ribbon uses is recycled so once the buffer gets full it frees the points at the back and uses that memory for new points.
PARTICLES

The particle class allows you to simulate simple particle systems that can appear visually stunning. The spawn location of the particles is determined by an ellipsoid where you specify the scale of the ellipsoid and the minimal distance to spawn the particles from (allowing you to spawn particles on only the surface of the ellipsoid if desired).

The particles have a velocity, acceleration, a scale, a billboard rotation, a particle index, and a colour tint. The colour tinting is implemented with a gradient class which provides great flexibility with colour – for example a particle can fade from white to yellow to red to black.

The particle index is done in a very clever way – all the different textures for the particles are placed on the one texture in a grid, so when you load the texture you specify how many rows and columns of particles that texture has. The engine will then generate the uv co-ordinates required to render the quads which have each texture clipped out. In the game the particle system is used to render the dust coming from the vehicles and the stars coming from the ghost. Due to alpha rendering requirements, particle systems that render with additive false (such as the dust) require the particles to be sorted so that the far away particles will render first.

The particle system provides two models of emitting – regular and oneshot. Regular means a value of 100 for emission means it emits 100 particles per second. Oneshot means the particle system will emit 100 particles once and then never again. The emission of particles in regular mode is well programmed so that if you spawn 20 particles in one timestep, each particle’s time will be determined realistically based on the previous timestep. This means if you have fast moving particles which emit at extremely high quantities you won’t get gaps forming due to inaccurate simulation. Likewise, the spawn position also uses this model so if you start at 0,0,0 and then move to 100,0,0 and spawn 100 particles, rather then them all spawning at 100,0,0 they will spawn between 0,0,0 and 100,0,0. This would be incredibly useful if the particle system was used to simulate a magic wand which followed the mouse and left a trail.

GRADIENT

The gradient class is a sophisticated gradient calculator which allows you to create advanced gradient maps like the ones used in painting programs. You can add a colour and an opacity, and then specify where in the ‘map’ you want it placed ranging from 0 to 1. A visual representation of it is this:

![Gradient Example](image)

An example of where we use this is in our particle system, where we allow a particle to change colours through its lifetime. For maximum performance a BinarySortTree search is used to calculate the colour
TRACK WAYPOINT SYSTEM

Our game uses a special waypoint system where each node is defined as a plane and a vehicle can only move forwards and backwards through the list of nodes. The plane is used to determine if the vehicle has gone past the node. If the vehicle was last in node 3 (ie. It had not yet gone past node 3) the system will test if it has gone through node 3. If it has, then it will take you to node 4 and test if you have gone past node 4 etc. Otherwise, it will test if you have gone backwards and possibly entered node 2, and if it has test node 1, etc. Through this system we can detect if the player is at the start of the track or at the end of the track, and it would also be possible to determine how much of the track they have completed.

REPLAY SYSTEM

The replay system used to drive the ghost was designed to take time into account. Rather than just store a snapshot of the vehicle each frame and play it back frame at a time, it defines how many frames to take each second. The reply system stores the time of each of these frames and stores them sorted in time order. When a replay is played, the current place in time is stored and as time progresses the replay can either go forward through the list (if time is positive) or backwards through the list (should the time be getting smaller, like the user wants to replay in reverse). The previous keyframe and the next keyframe are weighted together based on their distance from the target time to create an ‘inbetween frame’. This allows us our reply to work and be visually impressive no matter what the framerate is.
BITMAP FONT SYSTEM

Our game has a full bitmap font loading and rendering system, with a fast and flexible design. It allows the user to easily display and manipulate text on the screen, and benefits from a fast polygon based rendering system.

Features:

- Left, right and centre aligning text
- Ability to change the position origin (top left, bottom left and so on)
- Ability to set the position using pixels
- Ability to set the position using a percent relative to the screen
- Ability to set the size using pixels
- Ability to set the size as a percentage of the screen (relative size)
- Support for Character kerning
- String rotation
- Ability to set the centre of rotation
- Font colours
- Multi-line paragraphs of text
- Affectors that are applied to each character as they are drawn, for special effects
- Fully supports the AngleCode .fnt file format for loading fonts

The bitmap fonts in the font system are created using the AngleCode Bitmap Font Generator (freely available here: http://www.angelcode.com/products/bmfont/). This tool outputs a series of .png files and a .fnt file which describes how the font uses the .png files. Our BitmapFont class parses these files and loads them into memory, using display lists and quads to draw the characters to screen at an extremely fast rate.

The BitmapText class represents strings of text which have a position, rotation, font and so on. This class supports many features such as left, right and centre alignment; Setting the size and position of the text using pixel or relative screen size and so on. A full list is mentioned above. By separating the text class from the font class, we have decoupled the string from its graphical representation. It allows the user to load the font once, and use it with multiple strings of text.

Our font system allows us to set the size by pixels, or as a relative percentage of the screen size. This is a very powerful and easy system for positioning text. For example, to create a line of text which is always positioned in the middle of the screen, taking up 1/3 of the screen in its height, I simply set the size and positioning units to relative, the alignment to centre. And set the size as 0.33, and the position as 0.5, 0.5. This allows the exact same text appearance regardless of the window size.

One of the main goals while creating the bitmap font system was reusability. So far, we are yet to find a reliable, fast and easy text library for OpenGL; so it was extremely important that we could use this code in later projects. For this reason, the font system was particularly decoupled from most of our other code (the exception is the texture class). To render the text, you simply supply the draw function with the width and height of the window, and the font system does the rest. For these reasons it will be extremely easy to reuse the font system in subsequent projects.

Our font system uses functors called bitmap font character affectors. These are simple functor classes which are applied to every character as the text is rendered. Using character affectors we were able to create many visually stunning text effects which you see in the title menu and the game. The user can manipulate the OpenGL state before the final character is drawn, so whatever you can do in OpenGL you can also do in character affectors.
We chose to use texture-mapped bitmap fonts instead of geometric fonts for 2 main reasons. Texture-mapped fonts are extremely fast as they render one quad per character. Secondly, for our cartoon style game, we wanted to have very visually appealing fonts; this was not possible with geometric fonts. With bitmap fonts, we were able to export the font textures with a transparent background using AngleCode’s Bitmap Font Generator, load these into Photoshop and apply various layer styles and effects to the fonts.

POST PROCESSING

Our game contains a very easy to use post processing system. The post processing system is made up of 3 core components. The Shader class, the Frame Buffer Object (FBO) class and the PostProcessor class itself. Firstly, it is important to note that these 3 classes are all you need to move the post processing system to another project, so code reuse is very easy. The post processor works by storing 2 frame buffer objects. Firstly the scene is drawn to one of the FBOs. Then the that FBO is set as the current texture and drawn as a full screen quad to the other FBO. This is repeated, with a shader being set as active for each render, until all shaders (effects) have been rendered. At which point the final scene can then be drawn as a full screen quad by the post processing system.

The post processor itself is extremely simple to use:

```cpp
PostProcessor pp( screenW, screenH );
Shader* shader1 = new Shader();
shader1->loadVertexShader( "file.vert" );
shader1->loadFragmentShader( "file.frag" );
shader1->compileShader();
pp.addEffect( shader1 );

//Main game loop
pp.startRendering()
    //Render all the scene code here
pp.stopRendering();
//Draw the final screen
pp.drawProcessedScreen();
```

Each post processing effect is simply a GLSL shader, which allows the user full control over the post processing effect.

VEHICLE MVC PATTERN

Our game uses the Model-View-Controller design pattern to handle the vehicle data, controller and visual representation. By splitting the vehicle model (the physical state of the vehicle), the controller (input) and the view (how the vehicle is rendered based on the data) we were able to achieve an extremely flexible and reusable system which is easy to maintain.

Our game uses a VehicleKeyboardController to allow the player to control his vehicle using the keyboard, and simply by creating another vehicle with a VehicleReplayController we were able to implement a ghost which uses the same Vehicle view.

Our original game design involved an AI controlled vehicle which the player would be racing against. With the vehicle MVC pattern, we would simply create a VehicleAIController and attach it to a view, making it very easy to separate the AI from the physical and graphical representation of the vehicle.
VEHICLE SUSPENSION

We were able to very effectively fake realistic suspension for the vehicle by linearly interpolating between the old position and rotation of the vehicle, and the true physical representation. This is what gives the vehicle the floating look, as if it has real suspension.

TIME SCALING

To prove that all our code was written correctly to use time delta we implemented bullet time as a debug feature. It isn’t part of the gameplay, but instead just a nice way of seeing everything in slow motion to see whats going on with the particle system and the suspension etc.

All our code was written to incorporate timedelta. If we had a model like:
\[ \text{Speed} = \text{Speed} + \text{Acceleration} \]

It would become:
\[ \text{Speed} = \text{Speed} + \text{Acceleration} \times \text{dt} \]

And if we had a situation like:
\[ \text{Speed} = \text{Speed} \times 0.5; \]

We would implement it as
\[ \text{Speed} = \text{Speed} \times 0.5 ^ {\text{dt} \times 60} \]

Which would ensure that if dt was smaller or larger than expected that the physics would all behave correctly.

SOUND SYSTEM

We implemented an efficient and robust sound system using the OpenAL API and the ALUT library. This sound system is good in that it makes sure that a sound file is only loaded once, in order to conserve memory. It allows multiple sound sources then to share the same sound buffer. The sound system also supports fully 3-dimensional sound. The sound wraps up a sound file into a handy sound object using the Sound.cpp class. The SoundManager.h class then handles all instances of sound objects and also handles the playing, stopping and pausing of sounds.

ISSUES DOING THE MAINTENANCE PART

There were numerous problems with shays code as you are probably aware.

Firstly the units in his world are a very strange measurement. We did not want to create our world in these units so Dave worked out that 3.2 * shays units makes 1 unit equal to 1 cm. So after drawing shays world in our code we simply scale the world using glScale. And we then translated the world so the 0,0,0 coordinate was at the end of our corridor.

When we started drawing our models using display lists we noticed a strange bug where our models would be drawn in random locations. It turns out that this was because Shay manually called display list numbers, and out of the 507 that he had declared, not all of them were used, but were still being called. So when our models automatically generated a display list number, some would get a display list id below 507, and be drawn in shays drawing loop. To fix this a simple glGenLists(507) at the beginning of our program stopped our models from claiming any display lists below 507.
In order to model our world so it blended in with shay's world we needed to use his textures. But Shay’s textures were all .raw files. We were able to quickly dump all the textures to .png files using an OpenGL debugger named GLIntercept.

To handle shay's code better overall, we moved all his code into the “shay” namespace.

We needed to add extra bounding boxes to shay's world, but his bounding box classes were very badly designed, and increasing the amount of bounding boxes would result in a crash. So instead we decided to implement our bounding boxes separately from his bounding boxes. We did this by having shay's camera hold its own AABBWorld object (our bounding box manager) and checking on our own bounding boxes for collisions each frame. We did the same for plane collisions. We also needed to remove some of shays bounding boxes in order to extend our world, this mean sifting through his code and finding the correct bounding boxes to move.

The mouse movement in shay's world was severely broken, we fixed it by changing the glut mouse call back and changed the camera so that it stored x, y and z rotation instead of a look at point, this made it far nicer to work with in a first person shoot style view.

Shay's world did not have sliding collisions, so we fixed this by simply checking if we could move on each axis individually rather than as a whole. So if we could move on the z axis, we would move, if we could move on the x axis we would move. This was unlike shay's original function which checked both at the same time, so if either axis was being blocked the player would not move.

**SUGGESTIONS FOR IMPROVEMENTS**

**GAMEPLAY**

We had many ideas for different styles of gameplay but due to time limitations we weren’t able to explore them. Originally we had planned to implement AI so there were other vehicles on the track that you were racing against, and there would be powerups that you could collect. The powerups were going to either be defence, forward attack, or backwards attack. This would be implemented in a rock scissors paper style – forward attacks were things like missiles which could be stopped by defense shields, while backwards attacks were things like mines, oil slips (which would spin you round) and tar (which would slow you down) which would beat defense. Forward attacks would usually beat backwards attacks because the person behind can see who they are aiming at while people in front would be guessing where the other player is, also the people behind can see a backwards attack coming while the person in front would be unaware of a forward attack.

Not having time to implement Artificial Intelligence or a two player mode, we then considered some options for one player mode. Our first idea was a simple ‘collect the coins’ type system, where players would not only have to drive around the track but also collect all the coins before they could finish. This would add an extra element of challenge because if they missed a coin it would mean they could not complete the level and they would have to go back. However, we thought this might interfere with the flow of the game and would interfere with the drifting element of the game.

Another idea we thought about was having obstacles on the course that you had to destroy, which would be appropriate for a game titled ‘Minicar Madness’. The idea was that you would have a missile launcher on your car, and while driving around on the track there were obstacles in the way that you could either swerve around or shoot with your missile. The missile launcher would take a few seconds to recharge so the user would have to make the choice of whether it was better to shoot the obstacle and risk the next obstacle being even more challenging to get around, or attempt to get around the obstacle without shooting it and saving the missile for the next one.