

CGV Open Assessment

The Modelling of a Ferris Wheel

Paper Reference: 0620145

Exam Number: 51460

Table Of Contents:

1. Introduction	3
1.1 Textures Used.....	3
2. The Graphical Model	4
2.1 The Modelling of the Ferris Wheel.....	4
2.2 The Modelling of the Supports.....	4
2.2 The Modelling of the Supports.....	5
2.3 The Modelling of the Capsules	5
3. Animation of the Ferris Wheel.....	6
4. Different Viewpoints.....	6
4.1 Front, Side, and Above View	6
4.2 Capsule Views.....	7
5. The Landscape.....	8
6. Lighting	8
7. Conclusion.....	9
8. References	10
Appendix A – Class Diagram.....	11

Word Count: 2000 words

1. Introduction

The task for this open assessment was to model a rotating Ferris wheel, with a surrounding landscape, complete with nine different lights and a number of different view points using the MESA version of the OpenGL API. We were to write the program in C or C++, and were allowed to use any of their libraries, as long as the code compiled and ran on the department's PC's.

For the project, I decided to use C++ and restricted its use to Windows PC's only. This was because I wanted to use the Window's method of creating windows, and it's user interface, as it gives more freedom while programming. The project had five different units of work associated with it and they are: the modelling of the Ferris wheel, supports and capsules, the animation of the Ferris wheel, the setting of the different view points, the surrounding landscape, and the lighting.

1.1 Textures Used

A number of different textures were used in this project and a list of them can be found below. Most of the textures used were from OpenGL books or publicly available on the Internet, but some of them were created from a number of different textures by myself.

Splash Menu Textures:

textures/splash.bmp - Main splash menu texture
textures/buttonpressed.bmp - Pressed button texture
textures/startprogram.bmp - Texture for the start button clicked

Program Textures:

textures/ground.tga - The ground texture [1]
textures/steel.tga - Metal texture used in the capsules [6]
textures/windows.bmp - Texture used for the building's windows [5]
textures/rooftop.bmp - The texture for the plain rooftop
textures/helipad.tga - The texture for the helipad rooftop
textures/pooltop.tga - The texture for a roof with a pool

Skybox Textures:

textures/skybox/back.bmp - These textures are used for the sides of the skybox [4]
textures/skybox/bottom.bmp
textures/skybox/front.bmp
textures/skybox/left.bmp
textures/skybox/right.bmp
textures/skybox/top.bmp

2. The Graphical Model

I decided to use a 3d modelling program for the modelling of the Ferris wheel, supports and capsules. I found this to be the best method as you could see what your final model will look like in real time, and also because it incorporates all the data for the models within a couple of files. Furthermore, it makes your code cleaner as you don't have a large section that contains hard coded vertices and transformations.

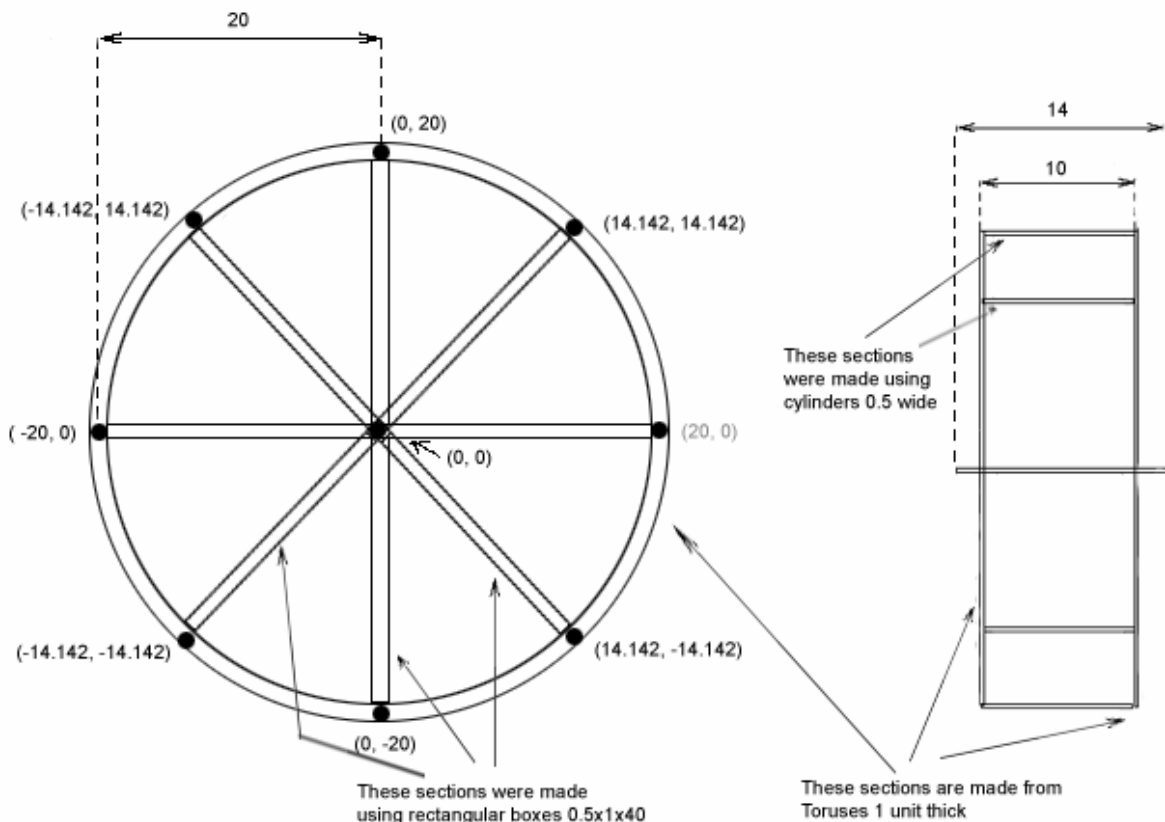
I used 3D Studio Max to model all of the components of the Ferris wheel. I decided not to use the 3ds file format to load in the models because the 3ds file format is rather awkward to use, and because I didn't have enough time to write a 3ds importer. Instead I imported the 3ds files into Milkshape 3D and then saved the models as a ms3d file. I then wrote a Milkshape model loader to load the models into the program (view Appendix B for the code).

The four models used in this project are:

- models/capsule-glass.ms3d* - data for the semi-transparent section of the capsule
- models/capsule-metal.ms3d* - floor and ceiling of a capsule
- models/wheel.ms3d* - data for the main section of the Ferris wheel
- models/support.ms3d* - data for a single support

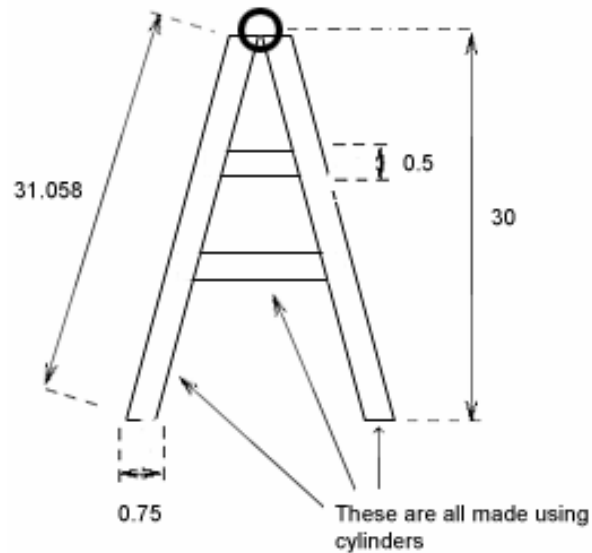
2.1 The Modelling of the Ferris Wheel

Since I was using 3ds max to draw the models, the modelling of the Ferris wheel wasn't that hard. Nevertheless, I did still need to know the coordinates of where the different components of the wheel should go. The following diagrams show these coordinates:



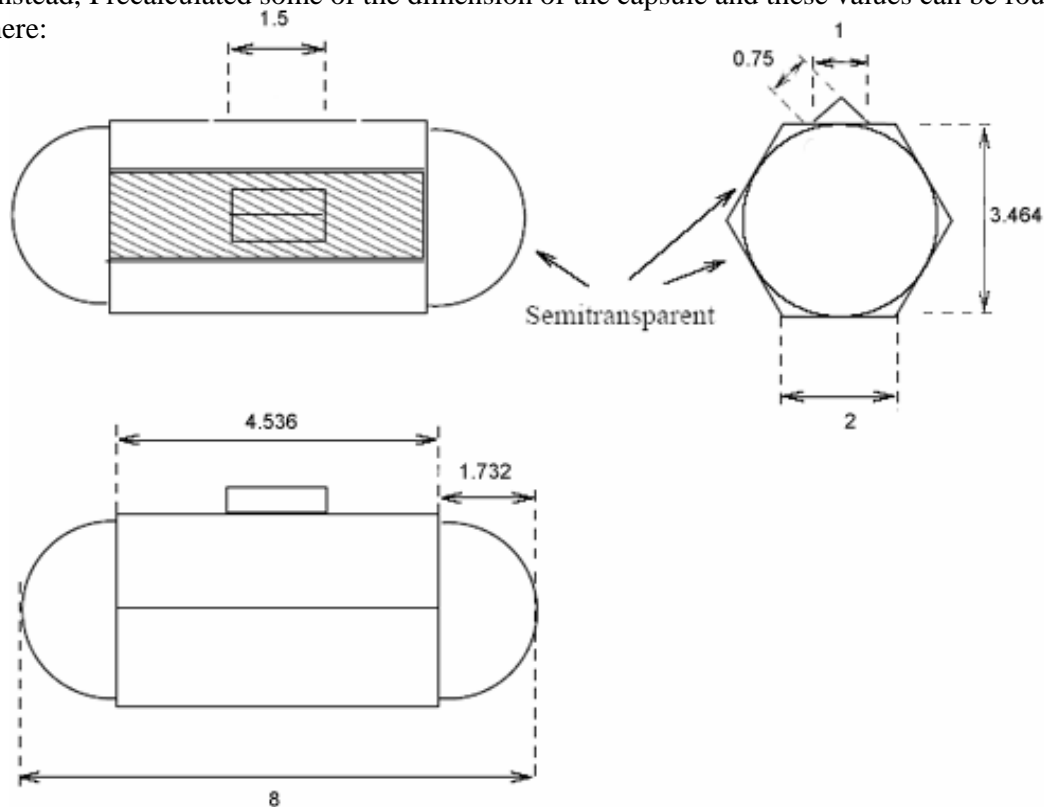
2.2 The Modelling of the Supports

The supports for the Ferris wheel were done in a similar way as the wheel. The following diagram shows some of the dimensions used:



2.3 The Modelling of the Capsules

I decided not to use the original capsules dimensions because it made the capsules look long and elongated and slightly out of proportion compared with the rest of the Ferris wheel. Instead, I recalculated some of the dimension of the capsule and these values can be found here:



The capsule model was split into two different sub-models: the metal section and the semi-transparent section. This was because of the way blending works i.e. all the polygons that need to be blended need to be rendered before the transparent section to get the correct results.

The modelling of both of these sub-models was relative straight forward. The only slightly tricky bit was the closing off of the section between the semi-spheres and the hexagonal prism, but this did not prove to be too big a problem since I was using 3ds max, and it contains some handy functions that allow you to create new polygons using existing vertices.

The rendering of the semi-transparent section of the capsule proved to be a problem at first as my original model loader couldn't deal with transparency but all that I needed to do was add a few lines of code in the `Draw()` function in the `MMEMilkshapeModel.cpp` file (see lines 260-265 & 284-288). Firstly, we needed to enable blending and disable depth testing. This allowed objects that are behind the blended section to show through the transparent sections. The second thing we needed to remember was that we needed to disable backface culling, as if it was enabled, we wouldn't be able to see the capsule when we were inside one of them (see lines 171-181 in `CGVWorld.cpp`).

3. Animation of the Ferris Wheel

There were two main tasks involved in getting the Ferris wheel to rotate properly. The first one was to get the main wheel model to rotate. This was rather easy in that a simple variable `zrot` was used to increase the amount of rotation each frame (the rotation was time-based so it didn't matter what the frame rate was). We then made a simple call, `glRotatef(zrot, 0.0f, 0.0f, 1.0f);`, before rendering the wheel to get the it to rotate.

The rotation of the capsules was slightly trickier in that they needed to stay vertical in relation to the wheel. Using `zrot` and some simple trigonometry, I was able to calculate where the capsules should be, and then translated each capsule to that location from the origin (lines 170-183 in `CGVWorld.cpp`).

4. Different Viewpoints

All the viewpoint changes were taken care of by a general Camera class (see `MMECamera.h` and `MMECamera.cpp` for the code). Input from the keyboard to change the camera mode was taken care of by the `CheckInput()` function in the `GLRenderSystem.cpp` file (see lines 148-180), and the drawing of and input received from menus was taken care of by the procedure handler, the `HandlePopupMenu()` and the `HandleContextMenu()` functions (see lines 270-365, 443-459 & 462-476 respectively in `GLRenderWindow.cpp`).

4.1 Front, Side, and Above View

All we needed to do to change to these viewpoints was set the camera mode to the appropriate mode (`FRONT_VIEW`, `SIDE_VIEW`, or `ABOVE_VIEW`). The update function in the camera class then made sure that the camera was positioned in the proper positions (lines 106, 110, 114 in `MMECamera.cpp`) (`PositionCamera()` takes care of updating the camera classes

member variables)), and also called the `gluLookAt ()` function with all the appropriate parameters.

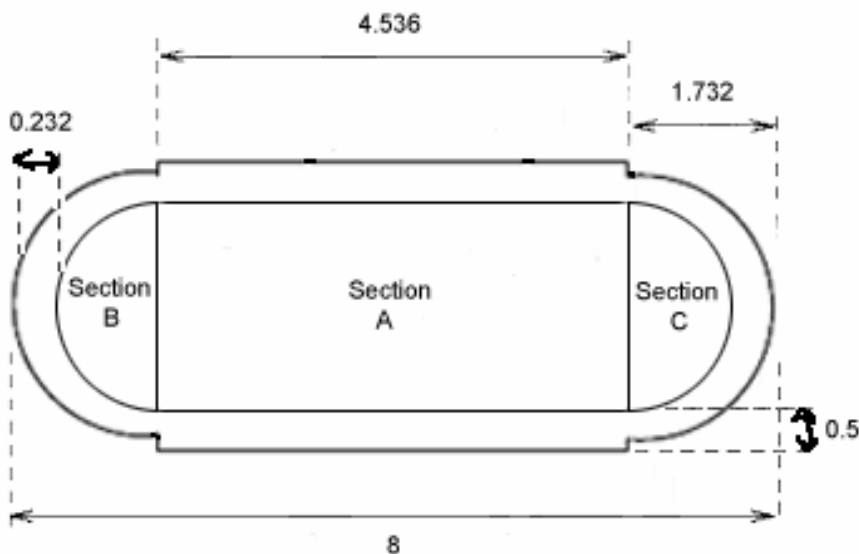
One thing worth noting is that I changed the front view so that it was slightly further away from the Ferris wheel. This was because when it was set to 30 units away, we only had a small portion of the Ferris wheel in our view. This was most probably due to the perspective settings I am using. Instead, I placed the view 50 units away from the middle of the Ferris wheel, which gives us a nice view of nearly the whole wheel.

4.2 Capsule Views

The capsule viewpoints were harder to implement in that the camera was constantly moving and thus we needed to update its position relative to the world to keep it in the same position inside the capsule. In addition to this, we needed to perform some collision detection with the walls of the capsules. Lines 118 & 124 in `MMECamera.cpp` take care of keeping the camera position in the same place in relation to the capsule it is in by storing the relative position in a static variable and then, at the next frame, updating the position of the camera before anything else is done.

After this, the first task is to check for movement through the `CheckForCapsuleMovement ()` function call. This function checks for input from the arrow keys and correspondingly moves or rotates the camera in the appropriate direction (see lines 286-321 in `MMECamera.cpp`).

Once we have received input from the keyboard, we call the `RestrictToCapsuleMovement ()` function (on line 121) to perform the collision detection with the walls of the capsules. The following diagram contains the values that were used for the different parts of the collision detection:



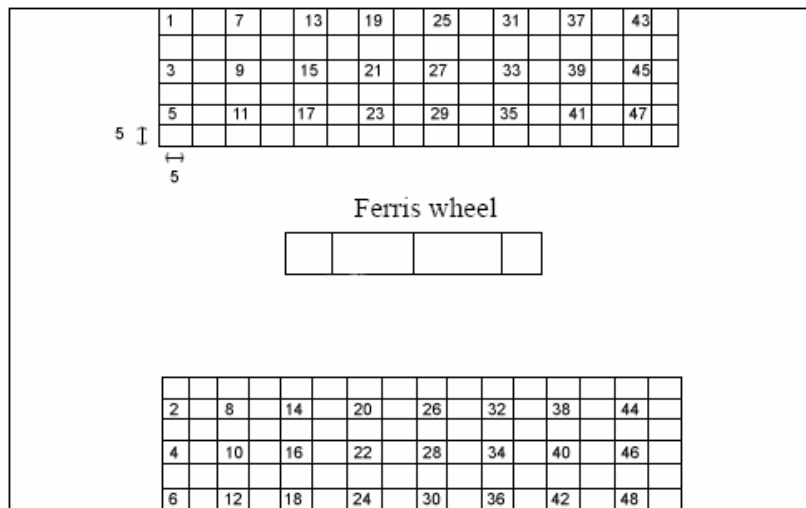
Collision detection for section A is taken care of by lines 326-330 in `MMECamera.cpp` file, but note that only the x-axis is restricted. This is because the z-axis collision detection is taken care of in sections B and C. The collision detection for these two sections is nearly identical and is implemented in lines 345-355 and 333-343 respectively. The way it is done

in the semi-spheres is by checking whether the current location is within a 1.5 unit radius from the middle of that sphere. If not, it calculates the location it should be by multiplying the actual location (with relation to the middle of the sphere) by a ratio that is 1.5 (the radius of the sphere we perform the collision detection on) over the distance from the center of the sphere.

5. The Landscape

The landscape was pretty straightforward to model. Basically, when the world is loaded in, the heights of all the buildings are initialised to be a uniform distribution in the interval [5, 25]. Once we have all the heights in the building data array, we loop through the array interchanging the current element with randomly selected one. This makes the heights of the buildings appear to be random (see lines 100-120 in CGVWorld.cpp).

The rendering of the buildings was then done in the following order by using two for loops (see lines 205-217 in CGVWorld.cpp):



6. Lighting

The lighting was one of the trickier parts of the project, in that it was tough to find good values for the different light components and attenuation values for the capsule lights. Lines 71-97 in CGVWorld.cpp take care of the initialisation of the lights. Since the capsule lights are static relative to the capsules, we need to update their position in the world each frame (see lines 191-198 in CGVWorld.cpp).

A point worth noting is that there are only lights for seven of the capsules. This is because OpenGL only supports eight lights per object, and thus we can only have a global light and seven capsule lights. One way we could have rendered all 8 capsule lights along with the global light would have been to use multipass rendering or vertex shaders, but these are overly both rather complex concepts and would have required a lot of time to implement.

7. Conclusion

There were quite a lot of other things I would have liked to mention in this report regarding the project (e.g. the way the windowing and texturing is done, the skybox, etc...) but as we have been limited to 2000 words, I am unable to do this. There are a couple of final things I would like to mention, and that is some of the improvements to the projects that I could have been done, if we had had more time:

- Environment mapping on the semi-transparent “glass” section of the capsules.
- Refraction of light coming through the “glass” section of the capsules.
- Shadows from the global and capsule lights.
- Realistic physics for the Ferris wheel.
- Frustum culling to not render objects outside of the frustum.
- The use of matrices to manipulate the location of object in the world.
- A node/object hierarchy for the models and other objects used in the project (this would help with frustum culling).
- Model manager, so I could have separate object instances for each of the capsules.

8. References

- [1] Dave Astle, Kevin Hawkins *OpenGL Game Programming (Game Development Series)*, Prima Publishing (2001), USA
- [2] MSDN Library – Windows User Interfaces
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/windowui.asp>
- [3] NeHe Productions – Tutorials 8, 13 & 31
<http://nehe.gamedev.net>
- [4] GameTutorials.com – Tutorials on Cameras, Transparency, Sky Boxes & Fonts
<http://www.gametutorials.com>
- [5] 3D Café – Free Window Textures
<http://www.3dcafe.com/asp/textureswindows1.asp>
- [6] nVidia's Transmogirfying Textures Volume 1
http://developer.nvidia.com/object/IO_TTVol_01.html

Appendix A – Class Diagram

