Medieval Knievel

Final Written Report

Presented to
Dr. Adam Krzyżak
Lecturer of Computer Graphics
COMP 371/2 LecUU
Concordia University

Prepared by
Group 8

Katrina Anderson	9106251
Christopher Di Fulvio	9614605
Patrick Modafferi	9401377
Matthew Tam	9675701

Undergraduate Students
COMP 371/2
LecUU

December 11th 2012

Table of Contents

List of Figures

List of Tables

1 Introduction

The objective of this project is to create a game using OpenGL and the principles and techniques learned throughout the semester in the COMP371 Graphics course. The game is called, Medieval Knievel, and consists of a player jousting against an AI on a motorcycle in honor of their king.

As a whole the members of this team had no graphics experience prior to this course; However all members of this team are well versed in object oriented programming and have formal training in the C++ language.

This document will describe our methodologies for the following program features:

1. User Interface Design & Robustness
2. Game Play & Logic
3. Texture Mapping
4. Lighting
5. Materials
6. Camera
7. Bike Modeling
8. Coliseum Modeling
9. Environment Modeling
10. Sound
11. Particles
12. Shadows

In addition, this document contains an installation guide and user manual to facilitate the running and manipulation of program features. Lastly, this document will explore the results of our project in terms of testing and discuss the extent of our learning during the project.

2 Methodology

2.1 User Interface Design & Robustness

2.1.1 Implementation

The user interface for the Medieval Knievel game is designed to interact with the player through the keyboard. More often than not, players will be presented with a list of their options along with the corresponding keyboard commands. In instances where they are not directly presented with a list, players can elect to view the instruction screen to refresh their memories on the various commands.

The Medieval Knievel game is designed around a series of game states: MAIN, BIKESELECT, GAME, FREE, CREDITS & INSTRUCTIONS. Each game state processes its own set of key inputs, which allows keys to have multiple functions and prevents users from giving commands that would be inappropriate in another given state. For instance, in the FREE state, users are able to control the camera position by pressing the arrow keys; they are not allowed to do so in MAIN state, where the focus must always be on the main menu. Key inputs also have controls, in the form of Booleans, which promote robustness and prevent players from pressing certain keys more than once. For instance, on the strategy screen, players must choose an integer from 1 to 4. Once they have chosen an integer they are no longer capable of pressing the integers 1 to 4 until they have entered the next phase of the game state.

Key inputs are advantageous as they allow users to give commands more quickly than if they had a menu and mouse system. The disadvantage to keyboard inputs is that some of them are not intuitive and are hard to remember and thus continued reference to the instruction screen is required. In the bike select screen, where the controls are limited, the lack of intuition is countered by implementing multiple view ports.

2.1.2 Alternative Implementations

Alternatively, we could have elected to create a combination menu and mouse user interface. This method of user input would be more intuitive, but much slower to input commands. In instances, such as the game animation sequence, where players may want to switch quickly between view points, this system would not perform well.

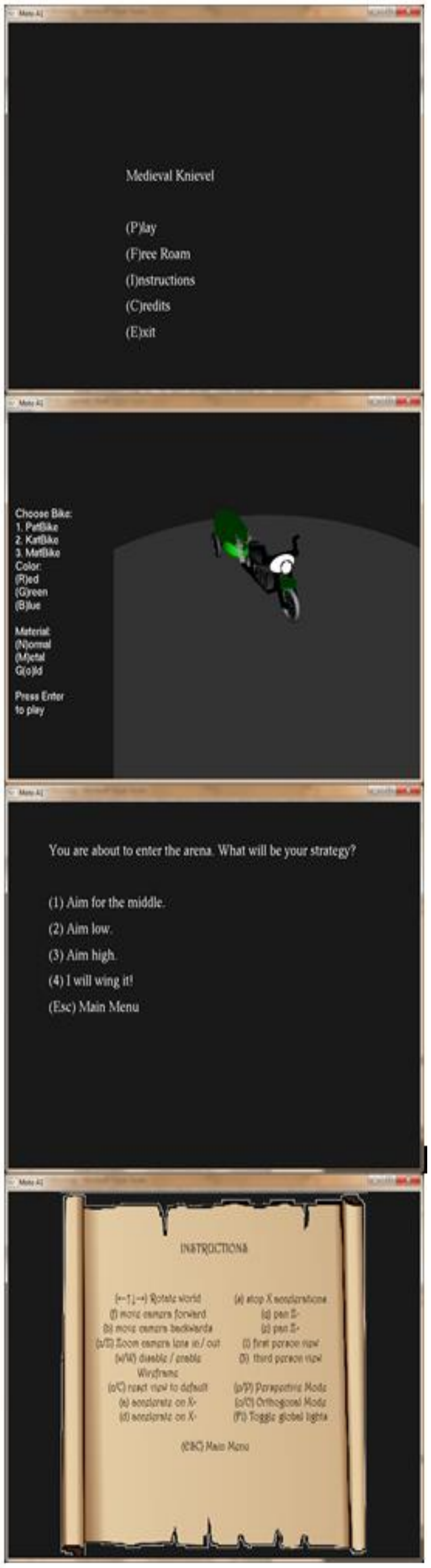


Figure 2-1 User Interfaces

2.2 Game Play & Logic

2.2.1 Our Implementation

In Medieval Knievel players face an enemy AI in a death defying motorcycle joust. This legendary joust takes place at night in an isolated arena inspired by the roman coliseum. Players are encouraged to customize their motorcycles and are required to choose a game strategy that will ensure their victory, result in a draw or cause their demise. Once players have chosen their strategy they can watch their fate unfold either from a first person perspective or from a third person perspective. For more detail on the bike selection and modeling aspects of Medieval Knievel please read sections 2.3 to 2.9 of this document.

Logically, Medieval Knievel is an advanced form of the popular children's game Rock, Paper, Scissors. Players have four strategy options, the last of which selects one of the first three at random. The AI for this game is relatively simplistic. It chooses one of the three options at random, with each option having a 33% chance of being chosen. A table of all possible outcomes is provided below. In an event of a tie, we give the advantage to the player just so that we can see the fading animation on one of the vehicles.

Table 2-1 Game Logic

Player Strategy	AI Strategy	Outcome
(1) Aim for the middle	(1) Aim for the middle	Draw
(2) Aim low	(1) Aim for the middle	Player wins
(3) Aim high	(1) Aim for the middle	AI wins
(1) Aim for the middle	(2) Aim low	AI wins
(2) Aim low	(2) Aim low	Draw
(3) Aim high	(2) Aim low	Player wins
(1) Aim for the middle	(3) Aim high	Player wins
(2) Aim low	(3) Aim high	AI wins
(3) Aim high	(3) Aim high	Draw

An advantage of implementing Medieval Knievel in this manor was that we were then able to create an animation that plays out according to previously provided parameters. We were also able to create a realistic AI. Developing this game in this way also gave us the opportunity to focus on more graphical aspects of the game as oppose to game play. As this project is for a graphics course, we felt the focus should be on the look and feel of the game as oppose to the game logic itself.

2.2.2 Alternative Implementations

Instead of implementing the joust with predetermined strategies, we could have elected to produce a game with collision detection and driving controls. This type of game would have been more realistic and possibly more fun to play, but would have taken a great deal longer to implement. A realistic AI in this type of game would also be time consuming to implement. Ultimately, other features of our games environment, like the advanced modeling of the coliseum as well as the particle effects, would have had to be sacrificed.

2.3 Texture Mapping

The project uses traditional texture mapping. This involves mapping textures using glTexCoord when drawing the different parts of the models. Since all of the models were made using OpenGL (none were made using any external software), the mapping needed to be done manually. For shapes that use quadrics, OpenGL offers functionality to automatically handle mapping. The functions used to map textures are housed within the texture manager class. The texture manager object is then passed to each model object.

Textures used were found from the website www.cgtextures.com. The textures were sometimes modified to fit what was needed for mapping specific parts of the models; For instance, the brick texture on the coliseum was modified because there were too many bricks in the original texture, and was cropped to look more genuine.

We also used a set of classes called imageloader.cpp and vec3f.cpp which were provided by the tutor, Kaustubha Mendhurwar, in our week 9 tutorial.

2.4 Lighting

Medieval Knievel was created using basic light sources throughout the application. The following subsections will detail the lights during each part of the application.

2.4.1 Bike Select

In Bike Select, there is a spotlight light from the camera which directs its light towards the motorcycle which spins in place.

2.4.2 Game / Free Play

During the game or free play, there is one light that emanates from the moon and illuminates the entire play field. This light can be toggled on and off by pressing the F1 key on the keyboard.

2.5 Materials

There are three materials, normal, gold and metallic, and three colors, red, green and blue, that are used solely on the bikes. The gold and metallic materials have a metallic quality, which is achieved by manipulating specular and shine attributes of the models. As for the color, some sections in the bikes have modified diffuse material properties that correspond with the bike’s RGB attributes. These materials are placed on the bikes during bike select and will continue to show into the game mode.

The shadows in Medieval Knievel have manipulated ambient and diffuse material properties set to black with an alpha property that allows them to be transparent. For more on shadows, please see section 2.12 of this document.

2.6 Camera

2.6.1 Our Implementation

The camera in Medieval Knievel is highly versatile and allows players to manipulate its settings to their liking. Players can chose for the camera to project in either a perspective mode or an orthogonal mode. In either mode they can rotate 360 degrees in both the X and Y directions. Players can also pan in either direction along the Z axis and physically move the camera forward or backwards along its line of sight. Players that wish to examine objects more closely or from farther away can do so by adjusting the camera's zoom functionality. Lastly, during the game, players can choose to watch the joust from a third or first person perspective.

While the camera is never limited in the Free Roam portion of the program, its focus during the game is targeted around the motorcycles and in the menu sections of the program, camera movement is prohibited. The advantage of our camera implementation is that it produces realistic views of our game, without letting the players lose themselves in changing perspectives.

2.6.2 Alternative Implementations

Our current implementation of the camera houses all the camera code inside the main class. This works as the code to manipulate the camera is small and readily available to all the game states. Alternatively, we could have separated the camera logic into its own class. Treating the camera as an object, follows more with the object oriented method, but it would also increase code coupling as each game state class would require one or more camera objects to be passed into it.

2.7 Bike Modeling

2.7.1 Our Implementation

Medieval Knievel features three motorcycle models: KatBike, PatBike and MatBike. These bike models were created and added to over the semester in a series of three assignments. For the Medieval Knievel game the three bike models are implemented in an object oriented way, where each bike model is represented by its own class that inherits from the parent "bike" class. The parent bike class ensures that each bike has essential attributes and methods that will allow the game to manipulate each bike in the same manner. Specifically, each bike model must have a draw, move and initialize method.

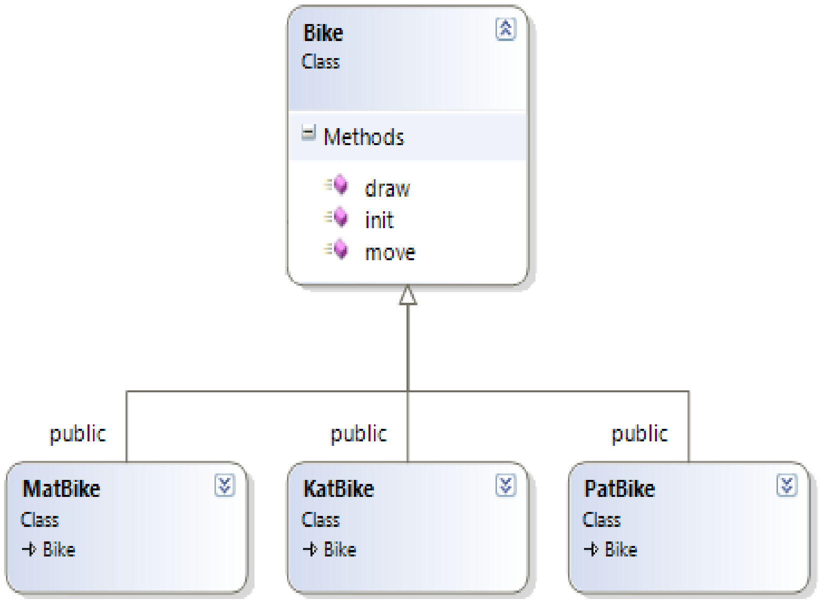


Figure 2-2 Bike Objects

2.8 Coliseum Modeling

Rome was not built in a day, but the coliseum model was, thanks to clever modeling and hierarchy. There are many ways to approach modeling in OpenGL, for the coliseum, we could have searched for pre made models or used external modeling software. In order to learn more about how modeling complex shapes works in OpenGL, our team opted to manually code, in great detail, this historical monument.

2.8.1 Research

First of all, we did some research on the coliseum and learnt that its façade is composed of two rings. The outer ring of 4 stories which is half broken-down due to an earthquake wraps the inner ring of 3 stories which is more or less intact. Obviously, it would be impossible to reproduce every detail of this giant structure so we will stick to the previously mentioned façade with its columns, and signature arches.

Figure 2-3 Night view of the coliseum we based our design on [8]

2.8.2 Planning

To model such a complex structure, a good practice is to break it down into hierarchical modules. At the base, the entire building is made of two rings. Each ring is made of floors, 3 for the inner and 4 for the outer. The floors are composed of a predetermined amount of arches which are accompanied by a single column.

See Appendix B for preliminary blue print sketch of the structure

2.8.3 Arches

To construct the arches, we first build two upright square prisms out of triangle strips for the walls and a single triangle strip at the top for the ceiling. Finally, the arch is placed at the center based on the two parameter x1 and x2 show in the Appendix B blueprint. These variables are helpful as we will see later since they offer flexibility in drawing various looking arches.

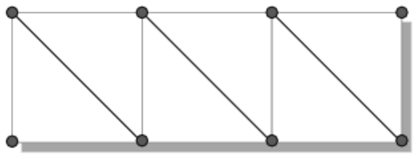


Figure 2-4 Example Triangle Strip

This last section is trickier; the curved arch must be accurately drawn without costing too much computation since we know it will be used nearly 300 times. The algorithm is a bit complex so we will try giving an overview of how it works. We think of our arch as having a front and back face. We can then render the remaining three surfaces (see Figure 2 5) by iterating back and forth on an angle from zero to PI three times, drawing one of the following triangle strips each time:

1- The curved arch surface:
   Maps vertices along a circular path connecting the front to the back face.

2- The front top part of the arch:
   Connects points on the front face along the circle from step 1 to parallel points on the ceiling.

3- The back top part of the arch:
   The same process as step 2 just on the back facing part so the structure looks nice from outside and inside.

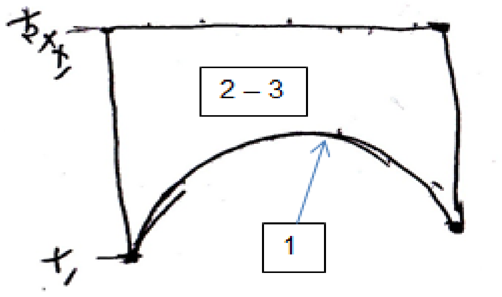


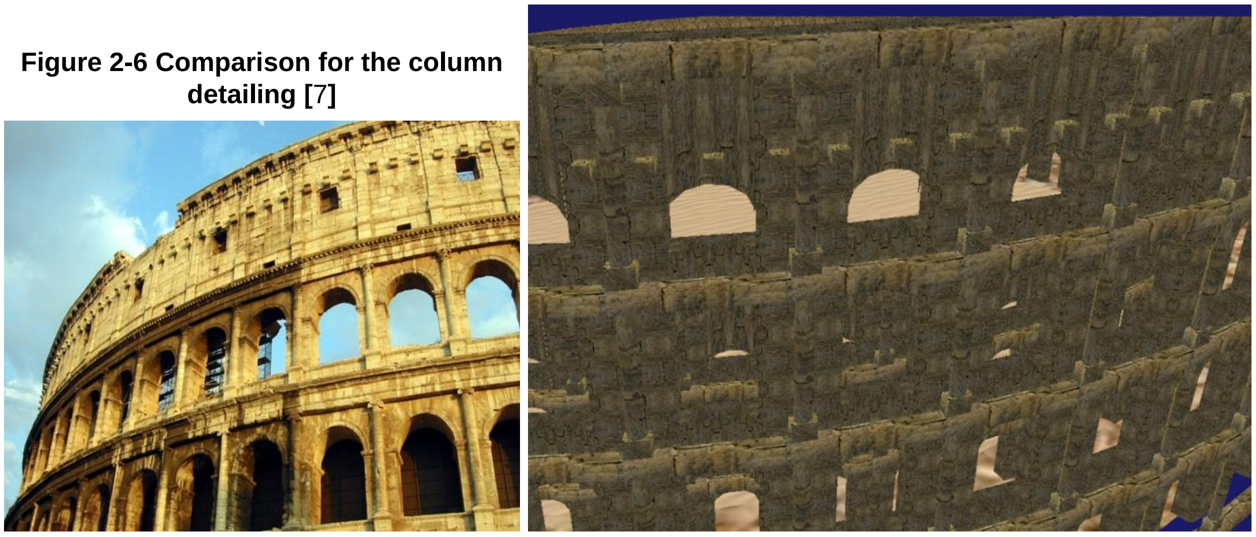
Figure 2-5 Arch sketch taken from APPENDIX B

2.8.4 Columns

Every time we make an arch, we place a column on a ledge in order to increase realism. The columns are a composition of built in OpenGL shapes. In fact, these columns serve more as a detailing for the building. They include everything from corbels, rims, blocks, beams and anything surrounding each arch. Most of it is trivial, however, to get a better definition and simpler texture mapping we opted to use cylinders with only four stacks and slices, and then rotated 45 degrees. We could have created more polygons out of triangle strips or quads, but the code would have become rather bulky.

For this component, we attributed a lot of importance to gaining a resemblance to the original thing due to the fact that the only purpose to this feature was to add realism. Below is a comparison with the real life model that we based the design on. Keep in mind the setting varies from day in the real thing to night in our version.

Figure 2-6 Comparison for the column detailing [7]



2.8.5 Floors/Ring of Arches

Each of the floors is built based on a starting degree and ending degree in radians. Typically, the start is 0 and the end is 2 * PI to create a full circle. The construction of the floor also receives the amount of arches as an input rather than a radius. This seems counter intuitive; however, if we reflect on the fact that setting a radius does not guarantee that the arches will meet when we want to create a full circle, we realize that we must compute the radius based on an amount of arches. The following mathematical formulae were used:

```
float step = 2 * PI / amountOfArches;
float ringRadius = (archWidth / 2) / tan(step / 2);
```

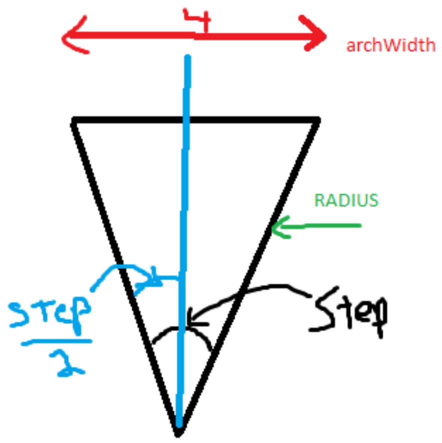


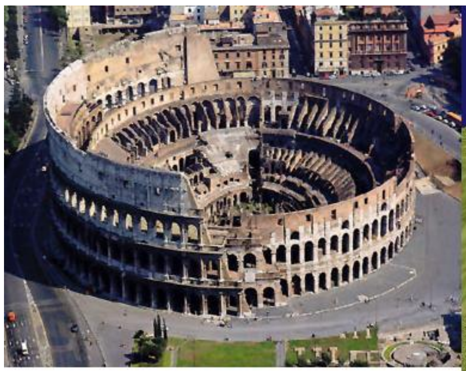
Figure 2-7 Geometry behind formula for finding radius

2.8.6 Walls

The roman coliseum is composed of two separate structures. In this case we will model the two walls that currently construct this landmark. The inner wall is the simplest part as it consists of 3 floors with the top one having modified arch starting variables, as mentioned earlier, ([x1, x2] => [0,6]). The outer wall is slightly more detailed; it is a larger version of the inner wall with a number of modifications.

1- The top row of arches is finished off with a special column which contains a top barrier and three protruding bricks as seen in the illustration above.

2- The start and end points gradually shrink with increasing height to reflect the crumbling of the real coliseum. For each layer, the ring of arches begins one unit later and ends one unit earlier.



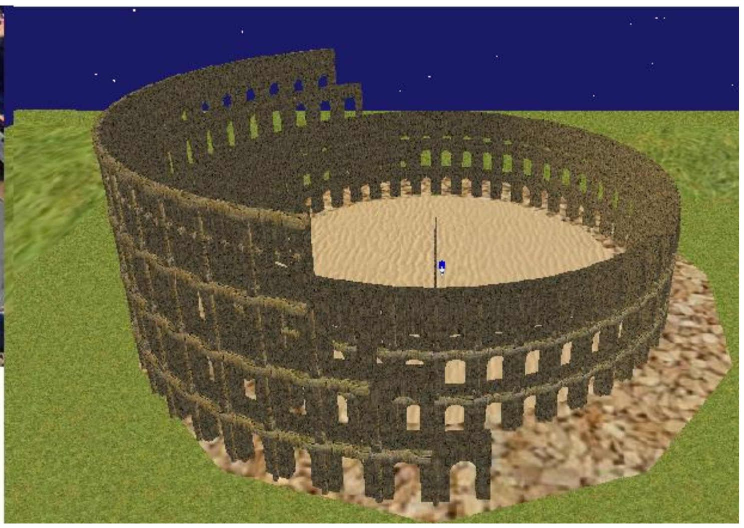


Figure 2-8 Comparison of the final product [9]

The end result if faithful to the current roman coliseum with its two signature rings, as seen in the illustration above.

2.9 Environment Modeling

2.9.1 Landscape

We will not go in too much detail since all the floors, the moon and landscape uses simple texture mapping explained in section 2.3.

The hills surrounding the coliseum are spheres that are scaled to look like hills. By stretching them in x and z direction then translating them downwards by slightly more than their y-radius, we get the illusion that the ground is undulated with hills from the Italian countryside.

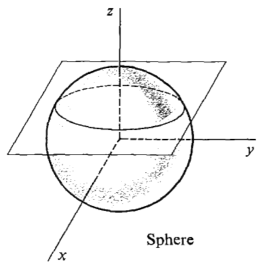


Figure 2-9 Sphere Intersecting with a plane [3]

For the fence, using the regular cube yielded some unwanted results. Therefore, to get optimal wood looking planks, we used GL_QUADS and manually mapped texture coordinates to vertices on the rectangular prisms.

2.9.2 Stars

Since the game is set at night, the sky is populated with an array of stars. Most people would have opted to find some texture for the sky to give the illusion of having stars all around. We thought it would be an interesting experiment to attempt creating physical particles that represent stars. We used a basic "struct" data type in C++ to model the stars. Each particle contains a pitch, yaw and scale. These values are all assigned randomly at initialization; yaw and pitch can be between 0 and 180. We then create a low definition sphere at the origin with 5 stacks and slices to give it a five sided look. They are then rotated by pitch and yaw around the x and y axis and translated in this new direction up into the sky. The material properties are purely specular and emissive white to replicate a star's shine or glow. Finally, if we back up the camera enough, we can observe all 200 stars placed randomly along a hemisphere. Each is unique in position and size.

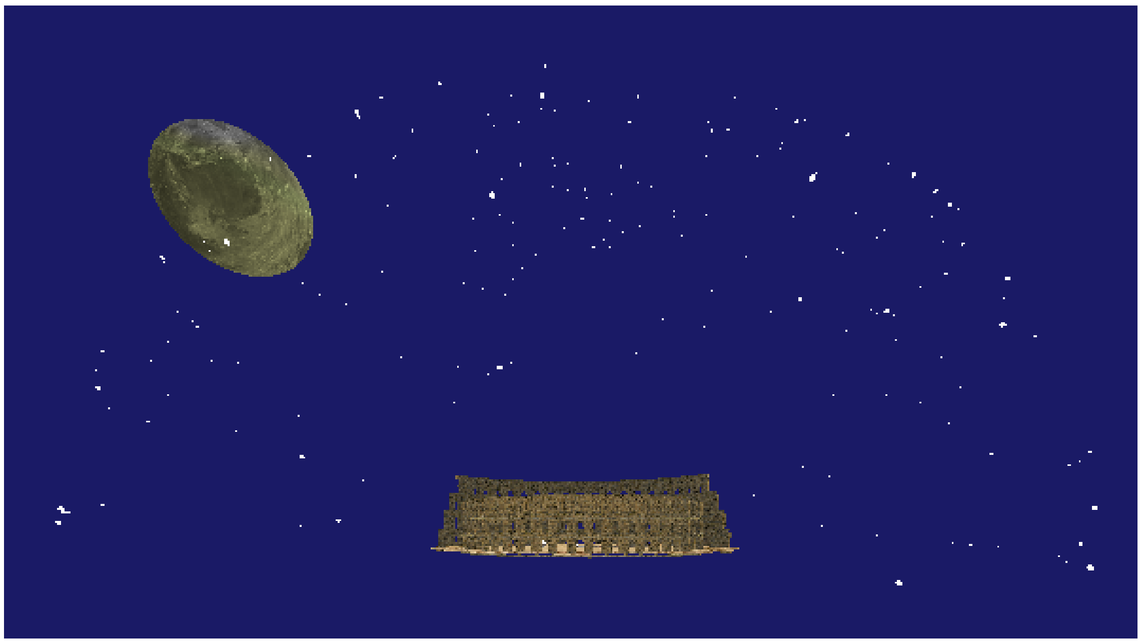


Figure 2-10 Screenshot showing dome shape of the stars

2.10 Sound

2.10.1 Our Implementation

In Medieval Knievel, music and sounds are used throughout the game to add to the game experience and to confirm when players have selected items from the menus. While browsing through the menus, players listen to a medieval tune and when they choose an option from the menu that will send them into another game mode, a unique fanfare confirms their decision. During the animation sequence of the Medieval Knievel game, players can hear the audio sequence of the motorcycles revving their engines as they accelerate, followed by the sound of their collision and the resultant skidding of their tires.

For sound in Medieval Knievel we decided to use the PlaySound function that comes with the Windows operating system. The PlaySound method can be invoked and controlled at any point in the program and has a minimal footprint as it is controlled by a single method call. The PlaySound method can play local windows sounds or any WAV file included with the project. The disadvantages of using the local windows library are that it doesn’t support sound mixing, so only one sound can be played at a time. In addition, sound will only play on the Windows operating system, reducing the portability of the game [**1**].

2.10.2 Alternative Implementations

Alternately, we could have included an outside resource or library as part of our project, such as OpenAL and SDL. These libraries support sound mixing and portability between operating systems. The disadvantage to using these libraries is that they often require additional installation on the player’s computer and time consuming configurations in project code. For this project we attempted to implement OpenAL and SDL without success and found their inclusion cumbersome in our project code [**2**] [**3**].

2.11 Particles

A particle system was used to generate the smoke emanating from the exhausts of each motorcycle. The way the particle system works as follows:

1. It iterates through every particle available in the system and makes them move in a random direction in the X, Y and Z directions.
2. It will then scale each particle and reduce its alpha every pass through.
3. Once the alpha of the particle becomes 0, we reset the particle back to its original position and redo the entire process for this particle.

The particles used in the final version of the application were tiny spheres of low slices. Originally, we were using very detailed spheres (10 slices) with a particle count of 1000. This caused a massive bottleneck when running the applications on PCs such as laptops or netbooks. Tweaking with the settings of the particles a little bit, it was possible to make the smoke look a little bit more realistic and drastically increase performance by reducing the amount of particles there are in the system. The reduction was from 1000 to 80 particles.

The particle system used was adapted from an already existing particle system code snippet found from the following website: http://snipplr.com/view/54306/opengl-particle-system-smoke-stream/ [**4**]

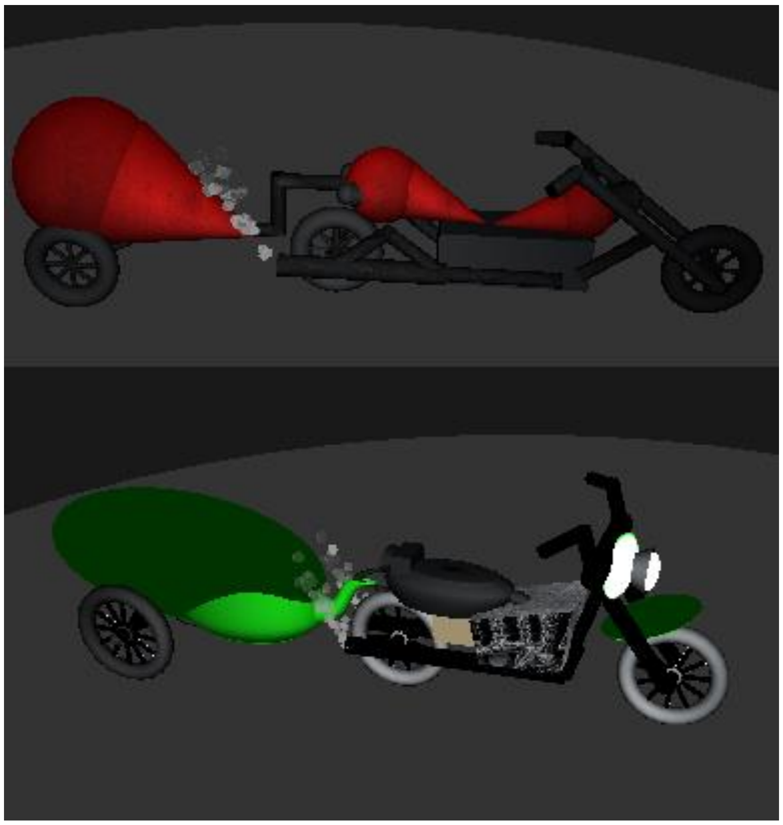


Figure 2-11 Particle Exhaust from KatBike and PatBike

2.12 Shadows

2.12.1 Our Implementation

The shadows in Medieval Knievel were implemented to solidify the position of the moon, the main light source, with respect to the coliseum. In the game's case, the moon is set slightly off to the side of the coliseum. So that shadows are created outside the coliseum, by the walls farthest from the moon, and inside the coliseum, by the walls closest to the moon. The shadows are multi-layered to represent the umbra and penumbra effect created in real life. This implementation of shadow is termed "hard," because of the sharp edges of the shadow shape. The advantages to creating a hard shadow are that they are quick to produce and more efficient to run. The disadvantage to this type of shadow is its lack of definition.

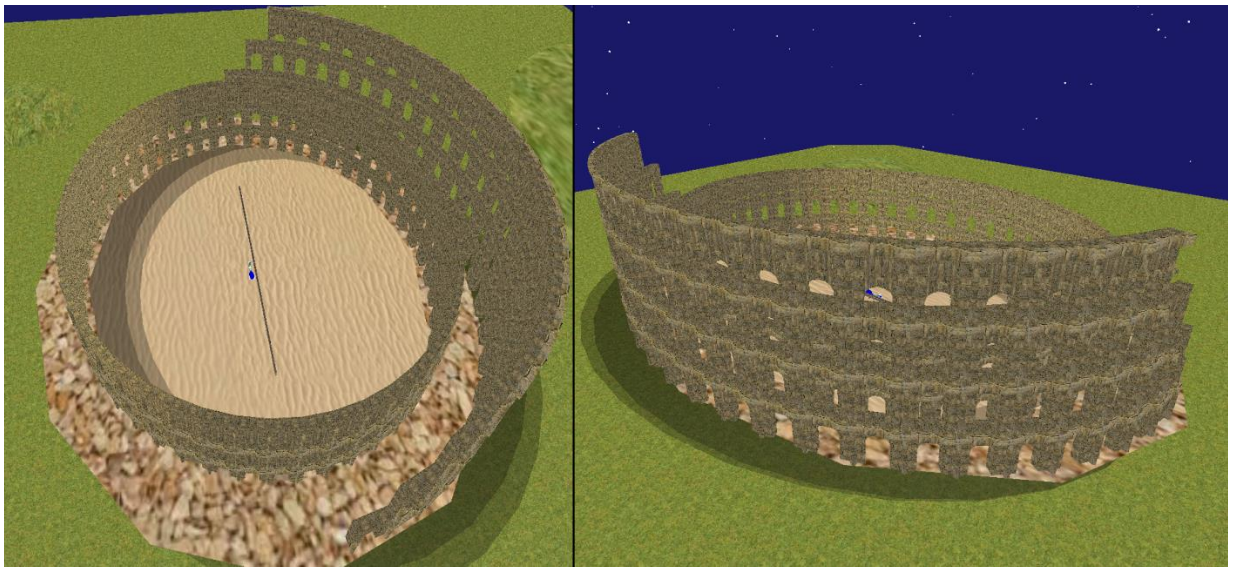


Figure 2-12 Shadows

2.12.2 Alternative Implementations

Alternatively, we could have created a shadow using OpenGL's stencil buffer. Stencil buffers are more difficult to implement and are more demanding on the CPU; however, the shadows created with stencil buffers have more definition than their counterparts and therefor look more realistic.

3 Installation Guide

The purpose of this guide is to break down the requirements and steps necessary to compile and run the Medieval Knievel game on a PC. Please note that there is more than one way to compile and execute Medieval Knievel, but the method outlined here, is the only one that has been tested by the developers. It is recommended that this program be run with the Windows operating system.

3.1 Required Libraries and Software

OpenGL (software is included as part of the Windows operating system)
OpenGL GLUT 3.7.6
Visual Studio 2010 Ultimate

3.2 Installing OpenGL GLUT 3.7.6 on Windows

1 Download glut-3.7.6-bin.zip from the OpenGL site
http://www.opengl.org/resources/libraries/glut/glut_downloads.php

2 The zip file included in step 1 contains the GLUT libraries and header files: glut32.h and glut32.lib and glut32.dll. Copy the files into the corresponding directories listed below, so that they may be referenced by Visual Studio projects [**5**].

Table 3-1 GLUT Directories

File	Directory
glut32.h	C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\include\GL
glut32.lib	C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\lib
glut32.dll	C:\Windows\System32

**Directories may vary from system to system*

3.3 Compiling and Running in Visual Studio 2010 Ultimate

1. If Visual Studio 2010 Ultimate or an alternative is not installed on your computer, please download and install from the Microsoft website:
http://www.microsoft.com/visualstudio/en-gb/products/2010-editions/ultimate [**6**].

2. Run Visual Studio and load the MedievalKnievel.sln file. To load the solution click on the “File” menu and select “open” navigate to the project submission folder and select the MedievalKnievel.sln file.

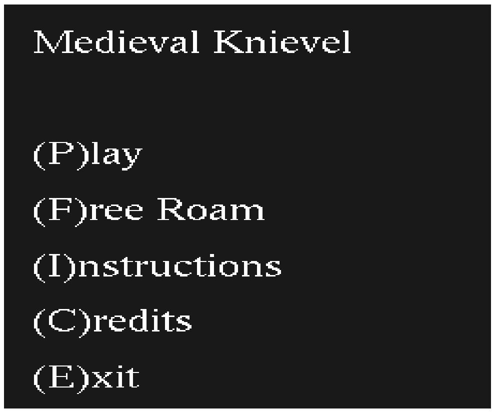
3. Compile the solution either by looking under the “Build” Menu and selecting “Compile” or by pressing Ctrl+F7.

4. To run the Medieval Knievel program select the “Run” option from the Visual Studio toolbar or press F5.

4 User Manual

4.1 Play Mode

To play the Medieval Knievel game select option "P," for play from the main menu.



Medieval Knievel

(P)lay

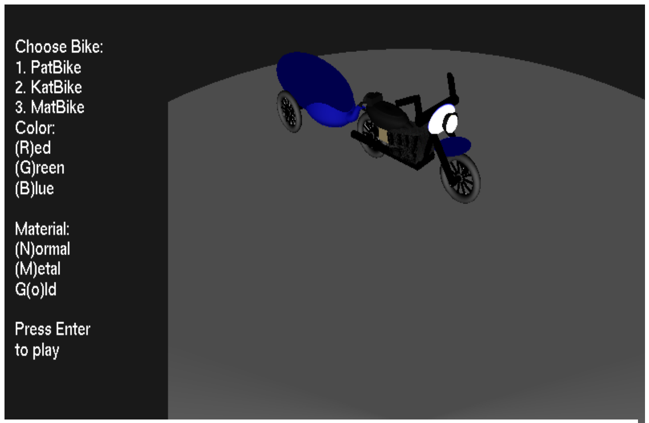
(F)ree Roam

(I)nstructions

(C)redits

(E)xit

Figure 4-1 Main Menu



Choose Bike:
1. PatBike
2. KatBike
3. MatBike
Color:
(R)ed
(G)reen
(B)lue

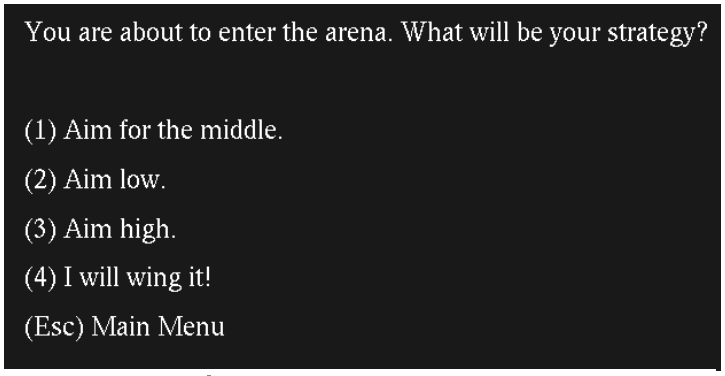
Material:
(N)ormal
(M)etal
G(o)ld

Press Enter
to play

Figure 4-2 Bike Selection Screen

Participants of the joust must equip themselves with a steed, which in this case is a motorcycle. You can choose from three bike models: PatBike, KatBike and MatBike, and have them painted in red, green or blue. Additionally, you may choose to embellish your motorcycle with silver or gold detailing.

Before entering the jousting arena, you must choose a jousting strategy. You can choose from the strategies listed in Figure 4-3. Picking to aim for the middle will result in a victory if your opponent aims high. Aiming low will pay off, if your opponent aims for the middle. Aiming high will beat an opponent who is aiming low. If you and your opponent joust with the same strategy, the result will be a draw.



You are about to enter the arena. What will be your strategy?

(1) Aim for the middle.

(2) Aim low.

(3) Aim high.

(4) I will wing it!

(Esc) Main Menu

Figure 4-3 Strategy Screen

Once you have chosen your steed and your strategy the resulting joust will be played out on the screen. You can choose to watch from a third person spectator perspective or from a first person jousting perspective, be pressing 3 or 1 respectively on the keyboard.

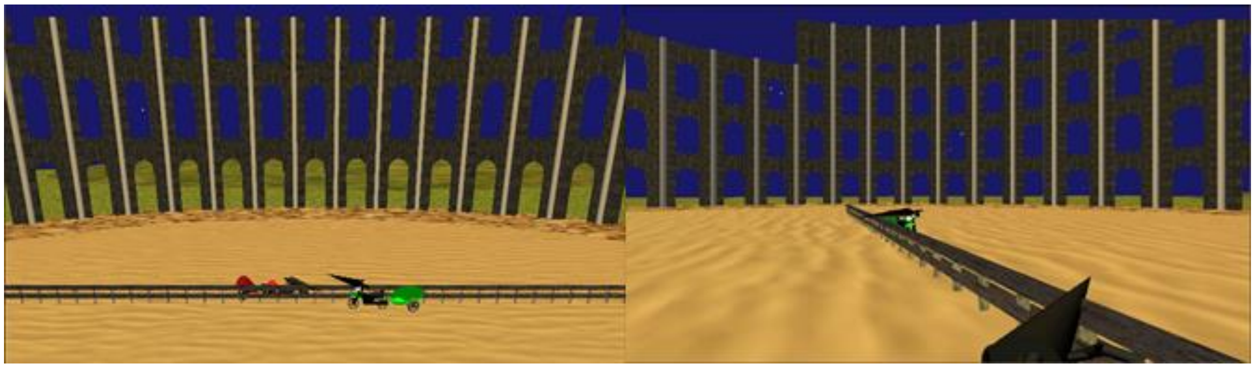
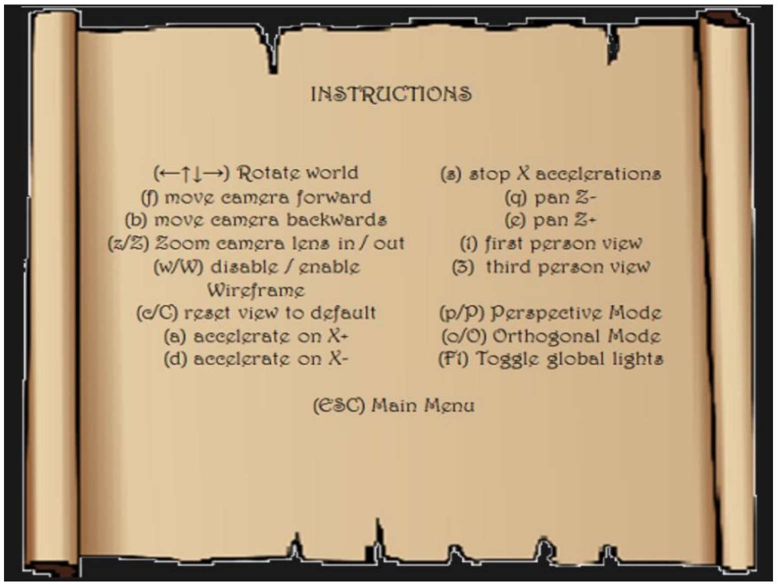


Figure 4-4 Game Play Third Person and First Person Views

4.2 Free Roam Mode

You can also choose to wonder the Medieval Knievel world by pressing “F,” for Free Roam, from the main menu (See Figure 4-1). In this mode you can use the controls listed on the instruction screen, see Figure 4-5 to change camera perspectives, lighting, bike movement and more…. The instruction screen can be accessed by pressing “I” at any point in the program.


INSTRUCTIONS

(←↑↓→) Rotate world
(f) move camera forward
(b) move camera backwards
(z/Z) Zoom camera lens in / out
(w/W) disable / enable Wireframe
(c/C) reset view to default
(a) accelerate on X+
(d) accelerate on X-

(s) stop X accelerations
(q) pan Z-
(e) pan Z+
(1) first person view
(3) third person view
(p/P) Perspective Mode
(o/O) Orthogonal Mode
(F1) Toggle global lights

(ESC) Main Menu

Figure 4-5 Instruction Screen

4.3 Credits & Exiting the Program

At any point during the game you may press the escape button to return to the game’s main menu, see Figure 4-1. From the main menu, you may exit the game by pressing “E.” It is also possible to see the credits for the game by selecting “C.”

5 Results (Outcome and Testing)

The team measured success by implementing new features that look great and made sure that we learned something while doing them. A more quantifiable approach to measuring success was taking these new features, and making sure that the application did not go below 30 fps on a top end PC.

There were two computers used for testing the fps difference for different implementations of models, particle systems and all other pieces of the application. The computer used before and for the demo is a decent laptop using an on board graphics card and an i5 processor. The tests done after the demo, were done on a desktop with an i5 processor and a discrete graphics card (GeForce 560Ti).

Any section with N/A means that it was not tested before the demo.

Item Name	Laptop FPS drop	Laptop FPS	Desktop FPS drop	Desktop FPS
Moon	1	49	0	60
Stars	3	47	0	60
Fence	1	49	0	60
Road	1	49	0	60
Coliseum	30	20	0	60
1 Bike	N/A	N/A	0	60
2 Bikes	N/A	N/A	0	60
1 Smoke	N/A	N/A	0	60
2 Smoke	N/A	N/A	0	60

Some improvements were made to specific components that allowed the game to be at a decent FPS for playability and smoothness (30 was our minimum on the desktop). The smoke was originally making a bottleneck in the application during gameplay, dropping the FPS to as low as 11 FPS. This was because the smoke that was being generated was very detailed (aprox. 1000 particles per stream of smoke, where each particle was a small sphere of slices = 10). To alleviate this, we removed a large amount of the particles and made the spheres a little bit bigger. The resulting smoke effect actually looked more realistic than before, as it was better defined and spread out. The new smoke uses only 80 particles at most, which did not even affect the FPS when testing on the desktop.

Our goal for the project was to keep the FPS above 30 when testing on the desktop. The results from our tests indicate that we have done a good job making sure that the models are generated with computer limits in mind. For example, the coliseum was generated mostly with triangle strips, which is very friendly to the GPU when trying to render.

6 Discussion & Conclusion

Overall, our team said we would consider our project a success if we completed the bike and environment models, bike movement, camera settings and gameplay. We managed to complete our original goal and more with special effects like particle systems, shadows, sounds and animations. During the construction of this project and the COMP371 course, we learned the theory and implementation of texture mapping, lighting, material properties, camera manipulation, object modeling, sounds, particles and shadows. We also gained valuable soft skills pertaining to team work and communication. Primarily, our team learned how to take a problem and break it down into pieces that can be scheduled and worked on efficiently. Given more time our team would have liked to contribute more models, possibly filling the coliseum with spectators and decorations.

APPENDIX A References

[1] Microsoft. (2012, December) PlaySound function (Windows). [Online]. http://msdn.microsoft.com/en-us/library/windows/desktop/dd743680(v=vs.85).aspx

[2] Creative Labs. (2012, December) OpenAL; Download. [Online]. http://connect.creativelabs.com/openal/Downloads/Forms/AllItems.aspx

[3] Sam Lantinga. (2012, December) Simple Directmedia Layer. [Online]. http://www.libsdl.org/

[4] Anonymous. (2012, December) www.snipplr.com. [Online]. http://snipplr.com/view/54306/opengl-particle-system-smoke-stream/

[5] The Khronos Group. (2012, December) OpenGL; Download. [Online]. http://www.opengl.org/resources/libraries/glut/glut_downloads.php

[6] Microsoft. (2012, December) Visual Studio 2010 Ultimate. [Online]. http://www.microsoft.com/visualstudio/en-gb/products/2010-editions/ultimate

[7] Aomarks. (2012, December) Wikipedia. [Online]. http://upload.wikimedia.org/wikipedia/commons/8/8c/Roman_Colosseum_With_Moon.jpg

[8] N/A. (2012, December) Destination 360. [Online]. http://www.destination360.com/europe/italy/rome/images/s/roman-colosseum-facts.jpg

[9] N/A. (2012, December) netvacations. [Online]. http://www.netvacations.ca/photos/attractions/31/31-2.jpg

APPENDIX B Arch Blueprint