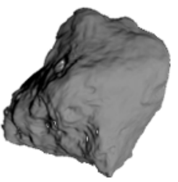
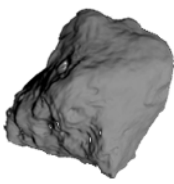
ASTEROIDS!
GALACTIC
SALVATION

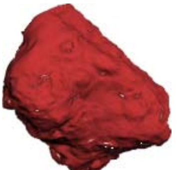
University of British Columbia
Faculty of Applied Science
Department of Electrical and Computer Engineering
EECE 478 – Computer Graphics

ASTEROIDS: GALACTIC SALVATION

Ryan Barr
Charanjit Dhanoya
Alexander Tse
Andrew Wei

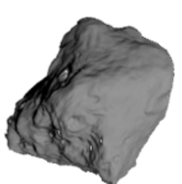






Abstract

The purpose of our game project was to create a 3D video game using OpenGL. We created a game similar to Asteroids, with extra features. The original Asteroids game was featured in 2-D and consisted of asteroids flying towards a spaceship controlled by the gamer, and the gamer had to shoot down these asteroids. Similarly, in our game, the user will have to shoot down asteroids but in 3-D space. We designed our game in C/C++ with OpenGL using the SDL (Simple Direct Media Layer) Library with features such as quaternions for full 3-D rotation and 3ds model loading for ships and asteroids.



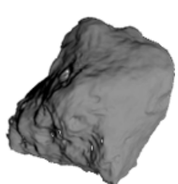
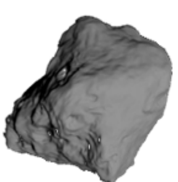
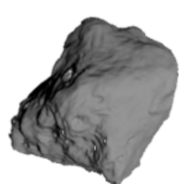
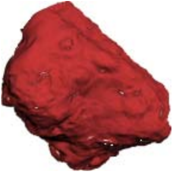


Table of Contents

Abstract 3
Table of Contents 4
List of Figures 5
List of Abbreviations 5
Introduction 6
Game Background and Objectives 7
Game Overview 8
  In-Game Features 8
  Technical Features 8
Game Controls 9
Technical Overview 10
Game Physics 11
Timing 12
Pausing 13
3DS Models 13
Importing Models 17
Textures Importation 17
SkySphere 20
Heads Up Display 21
Asteroids 21
  Asteroid Movement 22
  Asteroid Collision Detection 23
  Shooting 23
Conclusion 24





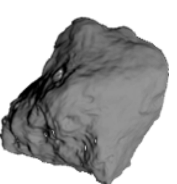


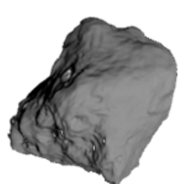
List of Figures

Figure 1 Pictorial Representation of Game Controls on the Keyboard 9
Figure 2 High Level Technical View of Asteroids Game 10
Figure 3 Quaternions using test cone as spaceship 12
Figure 4 Front view of spaceship model 14
Figure 5 Back view of spaceship model 14
Figure 6 Side view of spaceship model 15
Figure 7 Brown Asteroid Model 16
Figure 8 Red Asteroid Model 16
Figure 9 Grey Asteroid Model 17
Figure 10 Underside of Damiano Vitulli's ship model and with Asteroid models and HUD 18
Figure 11 Back view of Damiano Vitulli's ship model and with Asteroid models and HUD 19
Figure 12 Back view of our ship model and with Asteroid models and HUD 20

List of Abbreviations

2-D	**2 D**imensional
3-D	**3 D**imensional
3DS	**3D S**tudio Max open file format
HUD	**H**eads **U**p **D**isplay
GLUT	Open**GL U**tility **T**oolkit
OpenGL	**Open G**raphics **L**ibrary
SDL	**S**imple **D**irect Media **L**ayer

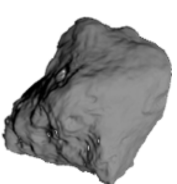


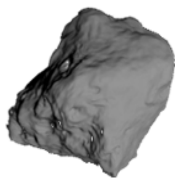


Introduction

This game manual style report describes the features of Asteroids: Galactic Salvation. The purpose of our game was to learn about game design but mainly create an awesome and entertaining game in OpenGL with SDL libraries. Because of the unique design of our game, we can offer full 3D rotation, trust and brake, as well an unlimited space for our ship to move as opposed to a confined space on Earth. While our game is based briefly on the original 2-D Asteroids game, it brings together many sci-fi entertainment shows and movies and combines religion as well. Asteroids: Galactic Salvation is a combination of Scientology, Battlestar Galactica, Terminator, and Total Recall.

This report will also satisfy the requirements of our EECE 478 Game project.





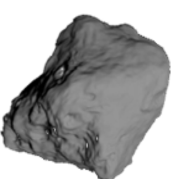
Game Background and Objectives

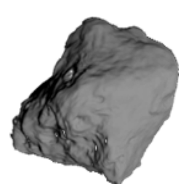
You are Cylon Model # 2, Leoben Conoy, and your belief in the one true god has been reinforced by the ultimate Galactic Confederacy of Xenu. Unfortunately, your creators believe in many gods with religions in Christianity, Islam, Judaism, Hinduism, Sikhism, Buddhism, Taoism, Bahá'í Faith, Confucianism, Jainism, Shinto, and Vodou. This not only violates you and your fellow Cylons' beliefs but also offends Xenu since he brought your creators to Teegeeack, now known as Earth.

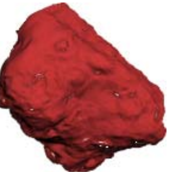
In order for humanity's children to flourish, their creators must die. Your base computer, Skynet, launches missiles at the Russian Federation, causing Mutually Assured Destruction on Earth. Earth blows up but takes out Venus as well and disrupts the flow of asteroids in the Asteroid Belt, thus, causing asteroids to drift in every direction.

A small group of 50,000 'body thetans' have managed to survive the attack and have escaped on roughly 60 space ships with the Battlestar Galactica leading the way and protecting the ships. They have semi-successfully terra-formed Mars into a mostly-livable environment and set up a base called New Earth there. However, chemicals in the Mars atmosphere have caused disfigurations in the thetans and without their Venus re-implantation station, their consciousness will not be implanted into a new body once they die.

This is a major advantage for you since Skynet built Resurrection Ships which you and your fellow Cylons use to once you die. Your objective is to take out as many asteroids as possible in order to reach New Earth to exterminate the rest of humanity.







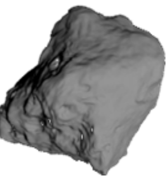
Game Overview

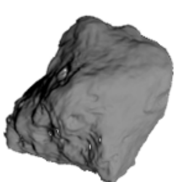
In-Game Features

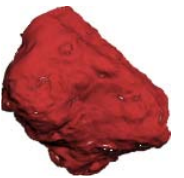
- 1 Level to shoot down as many asteroids as possible in your path
- Scorekeeping
- Health Monitor
- Limited Ammo that recharges over time
- Pause Menu

Technical Features

- Quaternions for 3-D rotation
- 3D Studio Max Model Loading & HUD elements
- Shooting
- Collision Detection
- Game Physics
- Randomized Enemy Movement
- 2nd/3rd person view



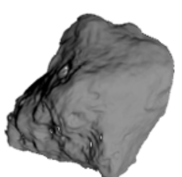


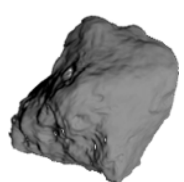


Game Controls

The player controls their spaceship with the keyboard. Pitch and yaw is controlled by a typical WASD setup. Forward and reverse thrusters are controlled by the J and K keys respectively. Holding down the L key kills translational velocity, and the spacebar is used to fire projectiles. Finally, the ESC key quits the game.

Figure 1 Pictorial Representation of Game Controls on the Keyboard

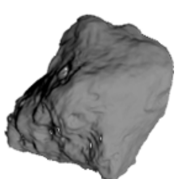


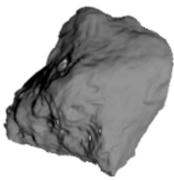


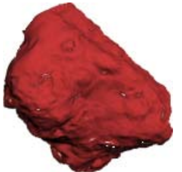
Technical Overview

Load 3DS Models
Initialize Game
Load Textures
User Input
Game Logic
Collision Detection
Render Scene
Orientate Camera
Render Skysphere
Render Objects
Render HUD

Figure 2 High Level Technical View of Asteroids Game







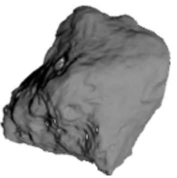
Game Physics

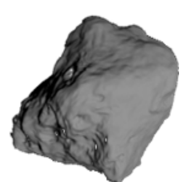
Since our game takes place in space, and allows all objects to move in any direction, we had to implement a simple Newtonian physics engine that handles acceleration and inertia.

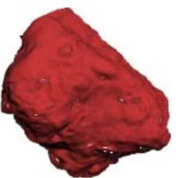
The first issue that was tackled was rotation in any direction. For simplicity, banking is not implemented, and may not be, as it is not needed to execute maneuvers in space. When implementing pitch and yaw, there was a problem with gimbal lock. If the player pitched or yawed 90 degrees, and then tried to yaw or pitch, respectively, they would instead bank in place. To fix this, quaternions had to be used. Instead of tracking three different glRotate functions, each object maintains its own 4X4 quaternion matrix that contains its rotation about an arbitrary axis. Thus, instead of calling the glRotate functions a simple glMultMatrix function is called.

Next, forward and reverse movement was added. This was tricky as the orientation of the object is stored in the quaternion as a rotation about an arbitrary axis. To solve this, two vectors were created, one for the x-axis and another for the y-axis (representing the pitch and yaw respectively) and each was passed through the objects quaternion matrix. Then the cross product of the rotated vectors was generated, thus providing the forward direction of the object.

Once this was completed, inertia was added to the objects. This required the storage of another vector, containing the direction of movement, which is now independent of the orientation of the object. Now when the object accelerates, the vector pointing in the direction of orientation is scaled by a value indicating the magnitude of acceleration and it is then added to the existing inertia vector. So accelerating in the opposite direction of inertia would slow down the object.







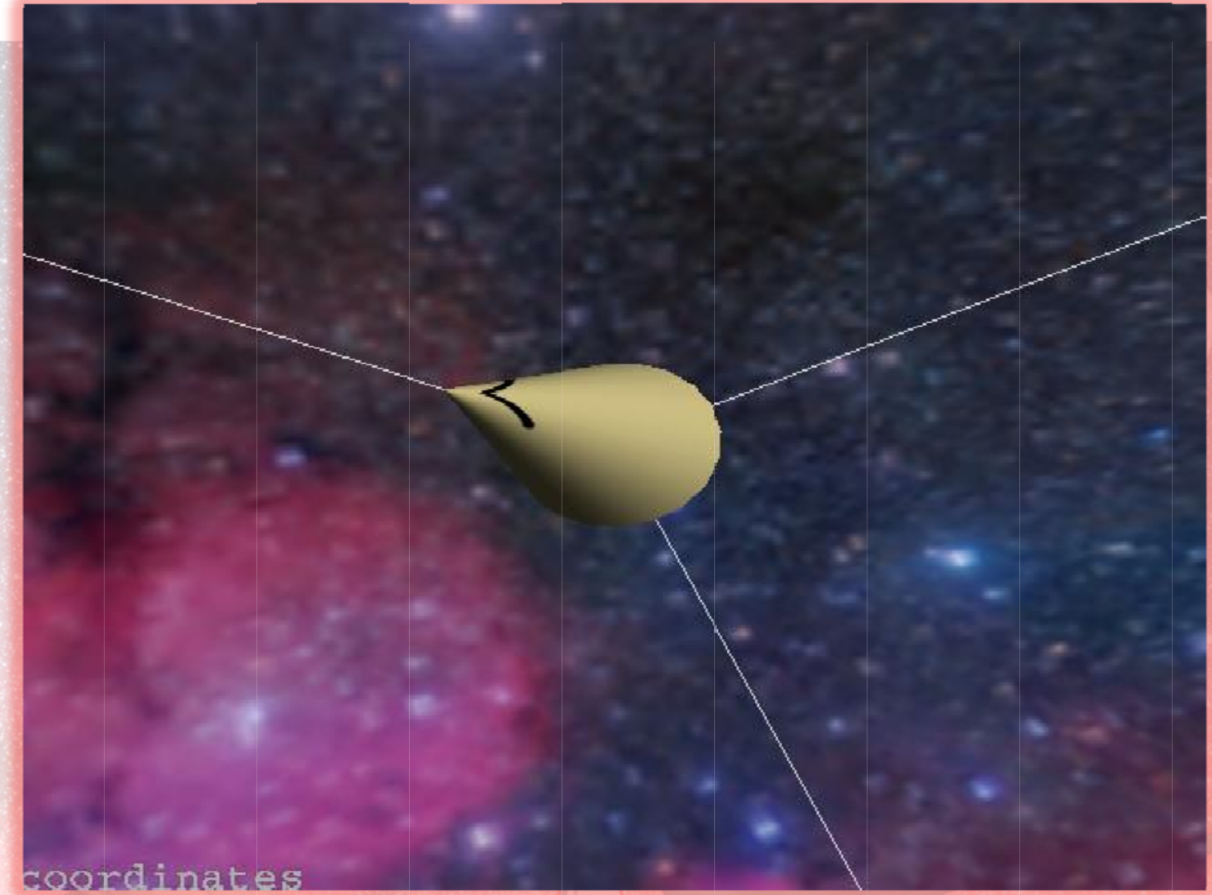
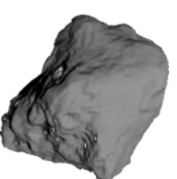


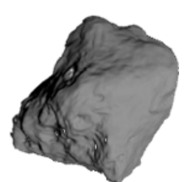
Figure 3 Quaternions using test cone as spaceship

Timing

To increase user enjoyment of the game, and to make it more realistic, we tied rendering to real time, and not to the frame rate. This way as the frame rate changes, especially on different computers, the rotation and translation of objects on the screen will remain at a constant rate.

We accomplish this by using the SDL_GetClicks function that gives us the milliseconds since SDL was initialized. The game keeps track of the previous value, and by subtracting the new value we get delta time. Delta time is then used throughout the program to multiply the rotational and translational





velocities of different objects, which results in a smooth rate of movement, independent of the frame rate.

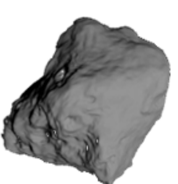
Pausing

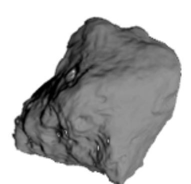
To implement pausing we simply lock delta time to 0, so nothing moves, and we skip all the code that allows the player to change pitch, yaw, etc. This allows the user to take a break from the game.

3DS Models

Within the game there are two main model categories: the spaceship and its surrounding asteroids. Initially, we decided to draw both models using simple objects in GLUT such as polygons and triangles. At first promising, this method eventually became a tedious process and the quality and complexity of the drawn objects are proven to be insufficient. We later realized the ability of importing external 3ds models into OpenGL, and as a result, all our finalized models are designed using the popular 3ds Max 2009 software from Autodesk.

The technique used in 3ds Max to produce our 3D models is basically starting fresh with a simple polygon and slowly sculpting the anatomy of the model. During this process, we took into consideration to not involve too much detail, thus reducing the number of vertices, which will in return optimize the overall speed of the game. For our spaceship, we started off with a Box object, set it as an Editable Poly, and began sculpting from that point on by repeatedly using Extrude, Bevel, and Insert options for editing polygons. Our final product is polished by adding the MeshSmooth modifier to soften the edges of the spaceship, and a material is chosen and textured onto the ship. Figure 4, 5, and 6 display our finalized spaceship in three different views: front, back, and side respectively.





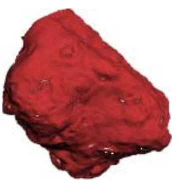
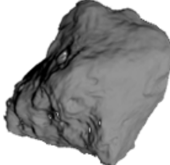
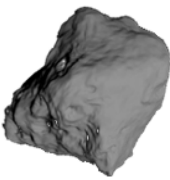
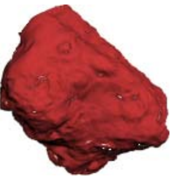


Figure 4 Front view of spaceship model

Figure 5 Back view of spaceship model







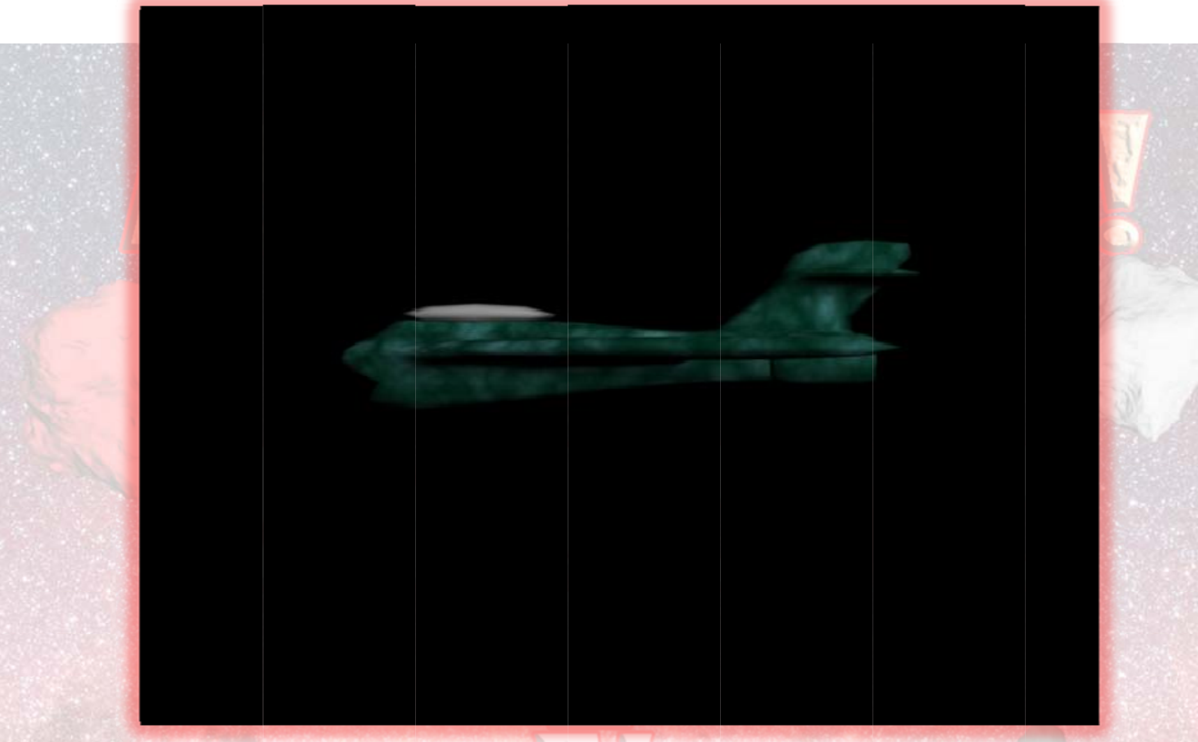
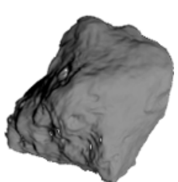
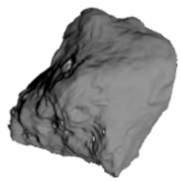


Figure 6 Side view of spaceship model

The asteroids were created in a similar way. Instead of starting with a Box object, however, the asteroids were made using Geosphere objects. To randomize the edges of the asteroid, the Noise modifier is repeatedly applied to achieve the desired effect. Several textures are mapped onto the surfaces to draw the craters. It is, however, too difficult to avoid having too many vertices in the asteroids. We created three versions of asteroids, basically similar, but with differently colored materials applied. Figures 7, 8, and 9 display our finalized version of the asteroids.





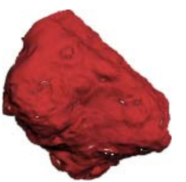
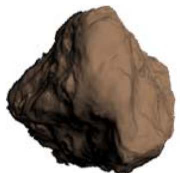
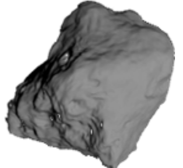
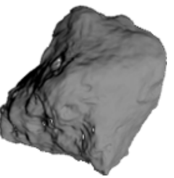



Figure 7 Brown Asteroid Model

Figure 8 Red Asteroid Model





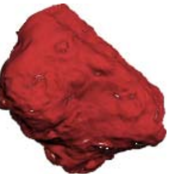
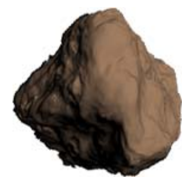
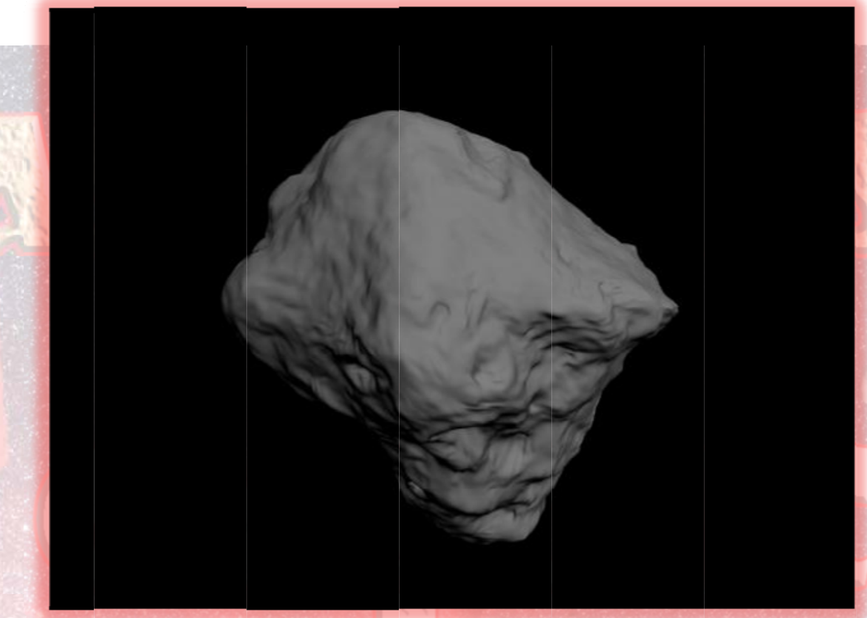




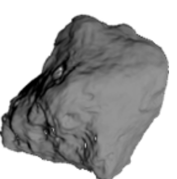
Figure 9 Grey Asteroid Model

Importing Models

Our program supports importing 3ds models for rendering. The 3ds file format is very popular and fairly easy to use. For our implementation we used source files written by Damiano Vitulli and posted on spacesimulater.net. This code provided easy integration, and loaded the textured models properly into our game.

Textures Importation

One of the reasons SDL was chosen is that it provides native bitmap support. For compatibility and ease of use we used Damiano Vitulli's implementation provided with his 3ds loader files. This works very well and allows for easy texturing of simple primitives and 3ds models.



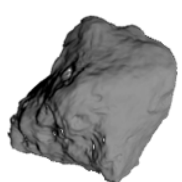
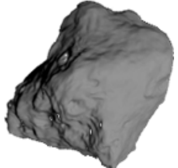
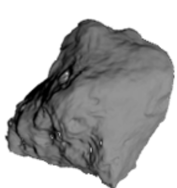
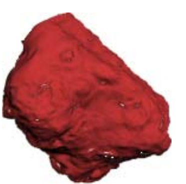


Figure 10 Underside of Damiano Vitulli's ship model and with Asteroid models and HUD







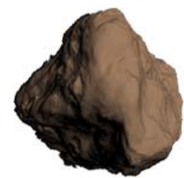
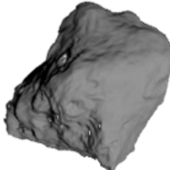
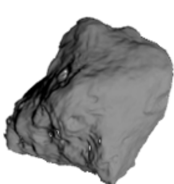


Figure 11 Back view of Damiano Vitulli's ship model and with Asteroid models and HUD





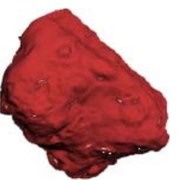
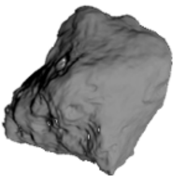
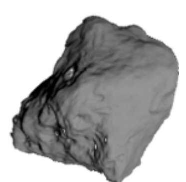


Figure 12 Back view of our ship model and with Asteroid models and HUD

SkySphere

Since our game takes place in space, we needed to implement a complete skysphere, as opposed to simpler skyboxes or skydomes. We implemented this by rendering a custom sphere. A texture made by Mike Pan at mpan3.homeip.net was used as it had built in distortion to counteract the distortion textures go through when being mapped to a sphere. Also by rendering a custom sphere it was easy to map the texture to it. Finally, during the rendering process the depth buffer is turned off, and the sphere is rendered first during the display process. This way everything else is rendered over top, and it appears that the skysphere is of infinity distance when it is actually rendered closer than some objects.



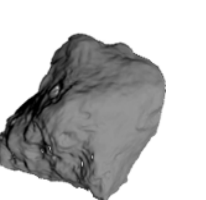


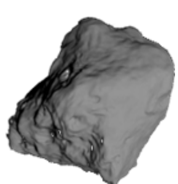
Heads Up Display

To help the player during the game a HUD is displayed showing critical information. At the end of the rendering process a new Ortho projection matrix is used, instead of the Projection one used for all other rendering, Depth buffers are turned off, and the HUD is drawn. This way the HUD elements always remain stationary despite how the camera moves. To draw the actual text we used texturing, and used source files written by Mark Kilgard. This method is one of the easiest to implement, is easy to position, scale and change fonts, and finally allows for colouring.

Asteroids

The number of asteroids is set in the game by level of game. Accordingly, that number of asteroids is initialized. The asteroid model is created in 3DS Max and the model is imported and appropriate initializations done. Each asteroid has a Boolean value called render, depending on which the asteroid will be rendered or not. The following flowchart graphically depicts the process



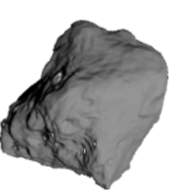


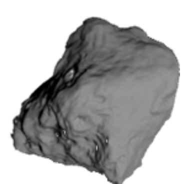
Start
Import 3DS model for asteroid
Create N asteroids
Initialize N asteroids and its attributes (position, speed, rotation)
Render?
TRUE
FALSE
Asteroids left?
TRUE
Render asteroid at ‘position’
FALSE
Update Level
End
Move asteroid based on ‘speed’ and ‘rotation’
Within World?
FALSE
Return to world
TRUE
Collision?
FALSE
TRUE
Other Asteroid?
TRUE
Redirect Asteroid
FALSE
FALSE
With Bullet?
TRUE
Render Collision Image
Update Score
Render [i] = false

The movement, collision detection of asteroids is explained below:

Asteroid Movement

The initial positions of the asteroids are generated in the program by rand() function and a few mathematical calculations based on the coordinates of the window. Each asteroid has attributes like speed, rotation. These attribute values are initialized to random values. This is also done in the asteroid initialization stage. For the movement of the asteroid, we keep updating the position value of asteroid





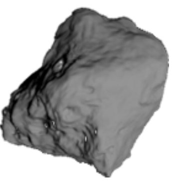
by rotating and translating it based on its rotation and speed attributes respectively. The motion of the asteroid is limited up to a certain x,y,z value after which the asteroid seems to disappear.

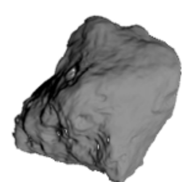
Asteroid Collision Detection

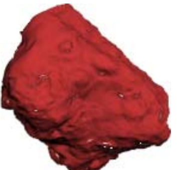
To perform collision detection between asteroids and/or between an asteroid and a bullet, we keep checking the distance between each asteroid and all the other asteroids and/or distance between each asteroid and bullets. For distance calculation, we assume that each asteroid and bullet is bound within a sphere of certain radius. The distance checking is then done by a module which calculates the 3d distance between the centers of the two objects, adding the radius of the bounding spheres. The result of the distance will determine if a collision occurred. If a collision seems to occur between asteroids, the asteroids are redirected by updating their positions. If a collision between a bullet and an asteroid occurs, we render a collision at that spot and then set the render Boolean variable of that asteroid to false.

Shooting

To implement shooting we initialize the bullet at the position of the spaceship. It would be exactly the value of the spaceship added with the radius of the bounding sphere. Also, during initialization, the speed of the bullet is also set to a predetermined value. When the bullet is shot, the module renders the bullet at the initial position and in the direction to be shot, which is orientation of the spaceship at time of firing. We use a linked list to track the bullets. It then is moved by continuously updating its position by performing translation based on the speed attribute of the bullet. As for the asteroids, the bullets seem to disappear after travelling a certain distance.







Conclusion

Congratulations!!! You've blasted through the Asteroid Belt. New Earth is almost in front of you. If you can blow all those ~60 ships away, New Earth will cease to exist. Unfortunately, you will have to wait for the expansion pack to fight off the Thetan ships.

Please stay up to date by checking out our website at www.AsteroidsGS.com.

To be continued....