Justin Ard
Alex Thomas
11/2/11
Project 2 Proposal

For our group project, we have chosen to implement an original space-based, action shooter game called Planet Defense (working title). This game will be written using the Java Binding for the OpenGL API (JOGL) libraries (http://jogamp.org/jogl/www/), and will be rendered using 3D graphics.

Game Overview

This game is a single-player, third-person (or first-person) shooter in which the user pilots a fighter spacecraft in 3-dimensions. As the title implies, the goal of the game is to defend your vulnerable home planet from the impending perils of hostile spacecraft and wayward asteroids. You must patrol the space surrounding your planet and shoot down any incoming threats using ship-board guns and missiles, while avoiding similar destruction yourself. But don't let down your guard, for your enemies are continuously emerging from the black recesses at the edge of space. The longer you survive, the more complex the threats become and the more frequent their arrival. Go for the high score by lasting as long as possible, until you are inevitably overrun and you, or your home world are destroyed.

Gameplay

As the game opens, you find yourself alone in space, orbiting your home planet. The game camera will alternately be positioned above and behind the ship (third person perspective), or at the front of the ship (first-person perspective). After an initial time period in which the user becomes acclimated to the navigational controls, asteroids and enemy ships will begin to appear at the distant boundaries of the game world (universe), and follow a collision course with your home planet.

After you have located and intercepted your target asteroid or enemy ship, you can destroy it by shooting it with your ship's guns. Some enemy ships will be equipped with similar weapons trained on you and your planet.

Your spacecraft and your home world will both sustain damage from collisions with asteroids, enemy craft, and enemy weapons. The amount of damage will vary, depending on the size of the asteroid/enemy craft and the type of weapon.

The user will control the spacecraft by using various combinations of key presses and/or mouse clicks. The primary navigation controls (yet to be mapped to specific keys) will be:

- Thrust (propel forward)
- Reverse thrust (stop forward movement)
- Bank (turn) left and right
- Climb (pitch upwards)
- Dive (pitch downwards)
- **Roll (rotate about longitudinal axis) left and right**[*]

Additional controls include those used for firing weapons:

- Fire main guns

---

* All high-bar optional game features are indicated in **bold** font.

- **Fire a craft-seeking missile**
- **Place proximity detonated mine**
- **Fire/use other specialized weapons acquired during the game**

The visual entities present in the game scene include:

- Your distant home planet
- Your spacecraft
- Enemy spacecraft
- Asteroids
- Background starscape
- Gun projectiles
- **Missile projectiles**
- **Proximity mines**
- **Additional space background: Moon, Sun, Galaxies, Nebulas, etc.**
- **Possible visual special effects such as explosions**
- **Cockpit with various heads up displays**

This game will continually challenge the user both physically and mentally, combining the quick reflexes, accuracy and timing demanded by a twitch-based action game, with the cerebral component of devising an effective defense strategy for warding off imminent doom.

Development Challenges

JOGL

As neither member of our team has previously written any substantial program in 3D, or even OpenGL, this project will necessarily have many potential sticking points. First and foremost of which will be learning to use the OpenGL and JOGL libraries while simultaneously creating a game from scratch.

Rendering

The rendering of the scene using OpenGL will have many unique challenges, many related to complex geometry calculations, including:

- Creating objects from primitive shapes such as points, lines, and polygons.
- Calculating normals for shading and collisions.
- Vertex and fragment lighting, coloring and shading.
- **Loading rendering data into vertex buffer objects for direct upload onto graphics hardware, yielding better performance.**
- **Applying textures and/or bumpmaps.**
- **Loading modeling data files for objects with complex geometries (.3ds, .obj, etc).**

Modeling Transformations

Because we are implementing our game in 3D, we will have additional geometry challenges related to accurately transforming all of the game objects to their correct positions. This involves various affine transformations such as scaling, translation and rotation. The complexity increases when you consider the number of coordinate spaces involved, including:

- World coordinates
- Individual object coordinates
- Camera coordinates

- Texture/Bumpmap coordinates

Collision Detection
We will also be faced with the familiar problem of collision detection and resolution between rigid bodies, with the added complexity of using 3D objects made up of polygons with many faces. This will most likely be another primary hurdle if we use objects with more than very basic geometries.

One simple solution to this problem would be to use basic bounding surfaces such as spheres or boxes for our collisions. However, if we want more realistic behavior from our game objects, we will again have to use complicated geometry to implement accurate collision detection.

If the user is skilled enough to remain alive for a significant amount of time, the world will eventually fill up with more and more game objects. This will ultimately lead to performance issues if we are constantly testing for collisions between all objects in our game world. This offers another opportunity for improvement on our high bar by implementing some form of optimization technique, such as organizing our objects using efficient data structures, like a k-d tree.

Collision Response
The level of complexity for collision resolution will depend on how realistic we want our game behavior to be. If we intend to simulate realistic physics for our high bar implementation, we are again faced with using complicated mathematics to compute the correct reflection direction and velocity after a collision. This will require us to simulate various physics principles such as center of mass, moment of inertia and angular momentum using Euler equations.

Timeline

In order to maintain sufficient progress throughout development, we plan to complete our game incrementally following the tentative schedule below.

Milestone	Tasks Involved		Estimated Date of Completion
	Task Description	**Team Member(s) Responsible**	
Alpha Milestone: • Camera positioned for third-person perspective • Visible spacecraft in 3D environment • Fully functional navigational controls • Functional ship-board guns • Space background rendered	Handle camera translations/rotations for third-person perspective	Justin	Friday 11/18/11
	Draw spacecraft	Alex	
	Handle Key/Mouse press events	Justin	
	Map navigational controls to movement in the designated direction.	Justin	
	Handle ship movement using 3D translations and rotations	Justin	
	Draw sphere representing home world (Possibly Textured)	Alex	
	Draw immobile background starscape	Alex	
Project Alpha Due			**Friday 11/18/11**
Beta Milestone: • Camera positioned for first-person perspective • Enemy ships visible and mobile • Asteroids visible and mobile • Collisions detected	Handle camera translations/rotations for first-person perspective	Justin	
	Draw enemy ships	Alex	Friday 11/25/11
	Draw asteroid	Alex	
	Program enemy movement	Alex	
	Program asteroid movement	Alex	
	Compute collision point	Justin	
Project 2 Updates			**Monday 11/28/11**
Release Milestone: • Collisions resolved • Completed graphics • Ability to restart the game • Point tracking • High score tracking • Visual Effects • Sound Effects • All controls and cheats implemented • Fully functioning game	Program collision response/resolution	Justin	12/7/11
	Add textures (optional)	Alex	
	Program End/Restart game functionality	Justin	
	Track scores	Alex	
	Display scores	Alex	
	Handel key press events for cheats	Justin	
	Program sound effects	Alex	
	Splash Screen	Alex	
Game Showcase	Program introductory tutorial	Justin	**Wed. 12/7/11**

Technical Showpieces

Showpiece 1: 3D Environment

Our first technical showpiece encompasses all the operations necessary to implement a dynamic, scrolling world in 3 dimensions. Because we are using the low-level openGL graphics libraries, rather than an off-the-shelf, high-level game engine, we must manually perform much of the complicated functions involved in maintaining the appropriate game and viewing conditions, and rendering the scene to the game window.

Simply creating the objects and putting them in their correct positions involves the following topics and operations:

- Modeling shapes with Polygonal Meshes
  - Calculating vertex and normal information
  - Assembling objects from geometry primitives
  - **Storing object information in vertex buffer objects to be loaded onto the graphics hardware**
- Transformations of objects with 9 degrees of freedom (df)
  - Translations in world coordinates (3 df)
  - Object rotations in local coordinates (3 df)
  - Object rotations in world coordinates (3 df)
- Three-dimensional viewing
  - Transformation of the camera with 6 df
    - Translations in world coordinates (3 df)
    - Rotations in world coordinates (3 df)
  - Perspective projections

Once we have correctly updated our game model, rendering our objects on the screen involves another set of complications, including:

- Creating light sources
- Creating shading models
- Hidden surface removal
- **Textures and bumpmaps**

Showpiece 2: Collisions

Since we are building our game from the ground up, we will also be implementing our own collision detection and resolution behavior. Different techniques will be used based on the types of game objects involved, including the following collision categories:

- Sphere collision detection using the line intersection test.
- Weapon collision detection using ray tracing.
- Convex polygon collision detection using the separating axis theorem.
  - Position of collision between moving bodies determined using the bisection method.

We will further improve our collision detection algorithm by applying the following optimization techniques and data structures:

- **Space partitioning of objects using a k-d tree.**
- **Course grain check for potential collisions using the plane sweep test on objects' bounding surfaces.**

High Bar

For the high-bar implementation of our game, there are many additional features and development techniques we may include. All features listed above in bold could be potential options for inclusion in our high-bar. Additional gameplay features might include:

- Bonus weapons/shields/powerups dispersed through the level or dropped by destroyed enemies.
- Requiring you to refuel your ship by picking up powerups dispersed throughout the game or dropped by enemies.
- Enemies using path-finding to maintain tactical positions when in proximity of the user's ship.

References

Resources which may be referenced in our game implementation include the following books and web pages.

- Rabin, Steve. *Introduction to Game Development: Second Edition*. Boston, MA: Course Technology, a part of Cengage Learning, 2010.
- Hill, F.S., Jr. and Stephen M. Kelley. *Computer Graphics using OpenGL: Third Edition*. New Jersey: Pearson Education, Inc., 2007.
- Moore, Andrew W. *An Introductory Tutorial on kd-Trees*. Carnegie Mellon University. <http://www.autonlab.org/autonweb/14665/version/2/part/5/data/moore-tutorial.pdf?branch=main&language=en>
- Chandran, Sharat. *Introduction to kd-trees*. University of Maryland Department of Computer Science. <http://www.cs.umd.edu/class/spring2002/cmsc420-0401/pbasic.pdf>
- http://www.geometrictools.com/
- http://nehe.gamedev.net/tutorial/collision_detection/17005/