COS 426 Spring 2009 Project
Deep Space Death

Johnnie Rose, Jr. and Chris Tralie

Goal: What did we try to do?

The idea was to create a basic 3D space flight simulator, controlled by a joystick, where the user flies freely through space and engages in firefights with some adversaries. The goal will always be to navigate through the environment and to destroy as many enemies as possible in a dogfight setting. It's been a while since any free space flight simulators have been released, so this would benefit anyone like Chris who is an X-Wing or Descent enthusiast.

Previous Work

Most of these algorithms have been done before, the purpose was to combine all of them into a game that's fun. In terms of 3D open space shooters, a few come to mind, such as X-Wing Alliance, Rogue Squadron, and Descent Freespace. The X-Wing series was extremely efficient on old hardware for describing large space scenes with many ships separated by large distances. The graphics on the older versions were very simple, but the gameplay outweighed this. Descent, on the other hand, was able to come up with very efficient lighting models for its time. And Descent 3 was notably efficient on older hardware for the graphics effects it had

---

High-Level Approach

Opengl is used for all of the rendering. We started with the codebase in rayview.cpp and R3Scene.cpp. We first added the capability for scene files to specify ships and objstacles, which, in addition to having an initial position, material, etc., also have a position and a 3D rotation orientation that can change with time. So we stored one matrix past the transformation matrix in the scene file at all times that could get them to this random position/orientation. Once we had all of this information, we could come up with a basic model where the obstacles and ships could evolve over time. This entailed registering a timer callback with opengl and doing timesteps with position, velocity, and acceleration. Roll, pitch, yaw are possible in the ships. It is also possible to switch between ships in the scene by hitting "C"

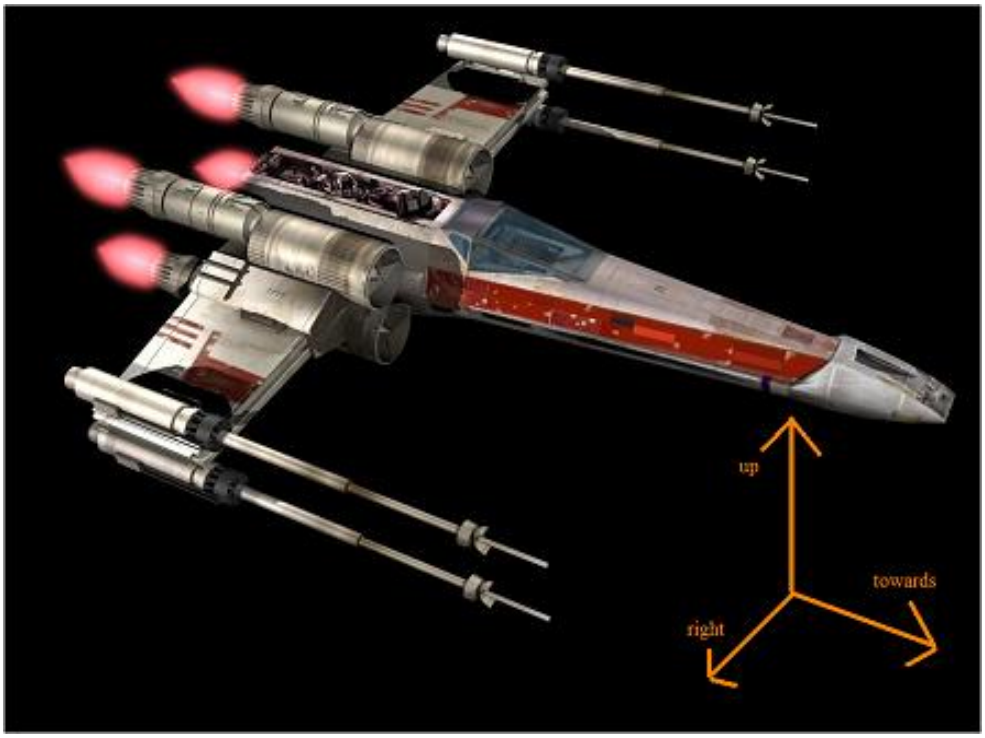
Once basic motion was done, we moved onto more complicated things. First, we came up with a basic weapons system (lasers) that uses ray-tracing to find collisions, along with a health/shields system for each ship (and health for each object). Then, an increasingly complex AI system was developed in parallel with some of the other effects, such as the cockpit interface and collisions with momentum response. For collisions, the damage done to each ship was proportional to the force imparted during the collision, and a realistic momentum response was attempted (after checking every object against every other object). Lastly, some semi-convincing, dynamic explosions were added with some basic sound when collisions took place, and dead ships/obstacles were removed from the scene.

These approaches will work well for a number of ships in the dozens and no more than 100 or so asteroids. This is because some of the algorithms don't scale well; collisions are n^2 (for n objects), as are a lot of the AI algorithm (each ship has to check every other ship and every other object to create its vector field, in addition to checking all n objects when it shoots rays out). So we are limited by speed because of some of the

algorithms we chose to use. But the system works very well right now if we adhere to these limitaions; it is possible to engate in firefights with enemies and for enemies to execute complex swarming behavior (keeping adequate distance between each other, darting in and out of asteroids, etc.). We know that it works well under these circumstances because we tested it a lot, and if we can keep the framerate down (by adhering to the limitations), the timesteps are small enough so that weird artifacts won't happen as much (like objects going through each other, etc.).

Methodology for Specific Tasks

Ship Modeling and Motion



For each ship's orientation, we stored an orthonormal basis of vectors **towards, right, and up**. To pitch, rotate the velocity vector and the towards/up vectors about the right axis. To yaw, rotate the velocity and towards/right vectors about the up vector. To roll, rotate the right and up vectors around the towards vector. When doing pitch and yaw, we check each time to make sure that the target rate does not exceed the maximum centripetal acceleration; that is, if a ship is going faster, it should not be able to pitch or yaw in as tight of a circle

We also created a function, **changeOrientation**, within the GameObject.cpp class that allows an AI ship to align itself with a given vector through a combination of rolls, pitches, and yaws.

Use the following matrix to transform an object from its local coordinates into world coordinates:

right_X	up_X	towards_X	position_X
right_Y	up_Y	towards_Y	position_Y
right_Z	up_Z	towards_Z	position_Z
0	0	0	1

Input and Control

User can either use the mouse or joystick. The joystick is accessed through the Windows API, so calibration is done externally. We mapped the mouse to pitch/yaw and the keyboard to roll and acceleration. The standard mapings for roll/pitch/yaw and thrust are made with the joystick.

Health and Weapons

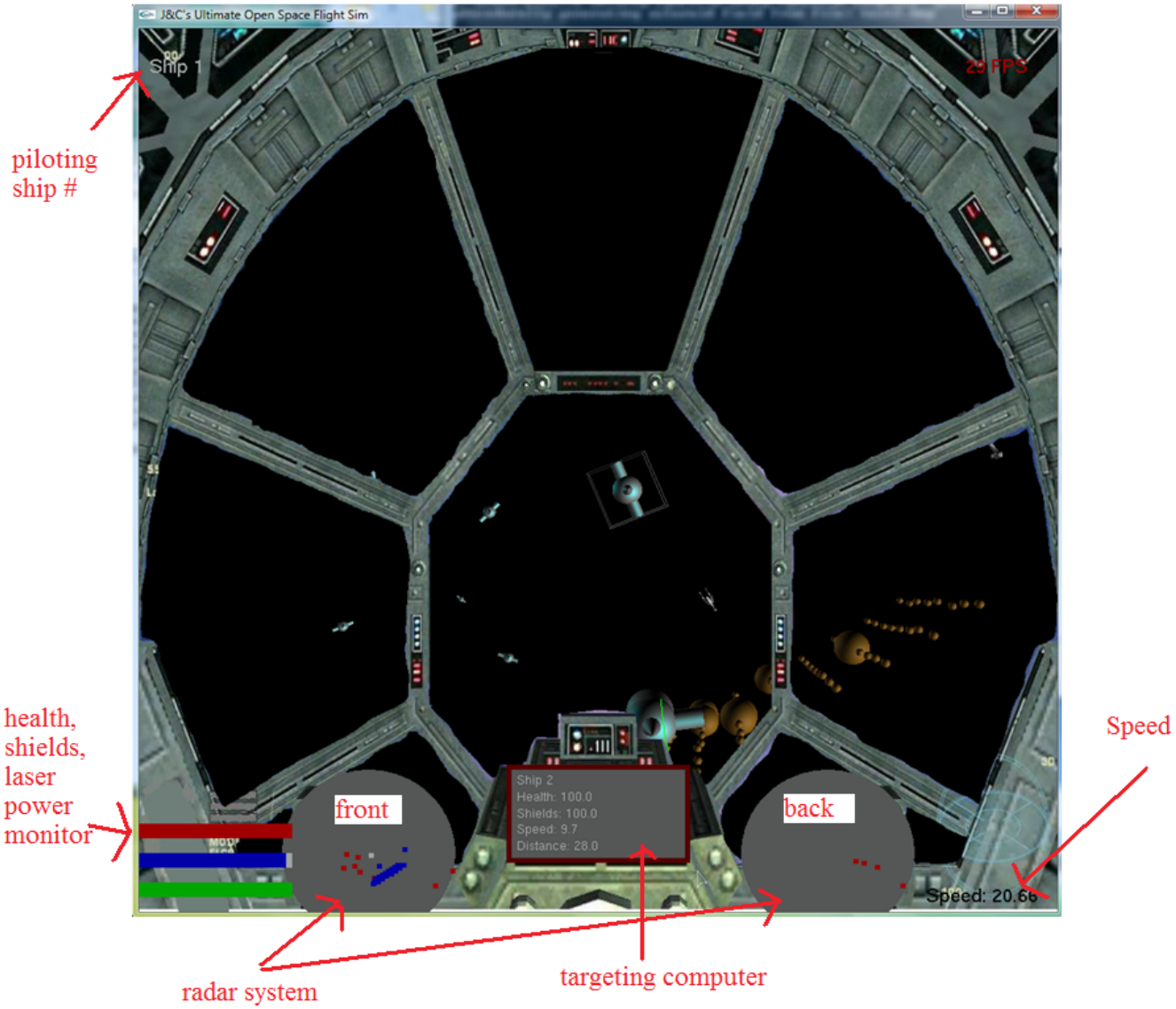
Each ship employs a laser cannon that finds its target, when fired, by casting a ray from the firing ship into space along that ship's towards direction. If an intersection with a positive t occurs, then the laser hit something---the amount of energy discharged by the laser in that timestep is passed onto its target as damage. Lasers automatically recharge using Euler integration: each timestep, a laser cannon recharges by an amount equal to the product of the laser recharge rate and the length of the current timestep.

When a laser beam hits another ship, the damage done is first applied to that ship's shields. Each ship is endowed with 100 shield points that, when decremented by laser fire or a collision, automatically increase using the same Euler integration method as implemented for the laser cannon. Any damage that exceeds a ship's shield points is then applied to that ship's health, which starts out at 100 points and only decreases over time---ships cannot regenerate health. To maximize realism, we give asteroids health but not shields so that

it's possible to fire upon or collide with an asteroid and see it explode.

Cockpit interface

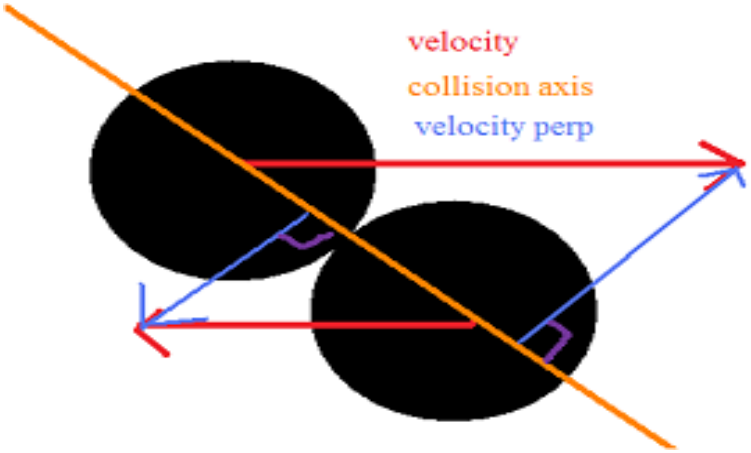
Create a cockpit for the player's ship to add another layer of realism. Do alpha blending so that the "windows" of the cockpit are see-through. Also, draw the outline of a bounding box around the ship that the player targeted.



Collisions

Each timestep, check every pair of objects to see if they collide. To do this, move one of the objects and then check to see if its axis-aligned bounding box overlaps the other object's axis-aligned bounding box. If this is so, then they will collide during this timestep, so move the first object back to its original position and update both objects' velocities based on the momentum response. Note that when asteroids are involved, a bounding sphere check is used instead since it's faster and more accurate in that case. Note also that this check is not 100% accurate but works well for our purposes at high speeds

If two objects collide, a physically realistic response needs to happen; we need to update the objects velocities based on the law of conservation of momentum. To make the equations simpler, we decided to resolve the momentum vectors into components parallel and perpendicular to the axis of collision (the line joining the center of the two objects). Momentum calculations can be done with the components along the axis of collision, but we preserve the components perpendicular to the axis of collision (we assume that they don't change). This implies that head-on collisions will be much worse than collision where two objects graze past each other on the size, since in the latter case the component along the axis of collision is much smaller.



Now we just apply the 1D formulas for the momentum responses along the axis of collision, with a random coefficient of restitution between 0 and 1.

The formulas for the velocities after a one-dimensional collision are:

$$V_{1f} = \frac{(C_R + 1)M_2V_2 + V_1(M_1 - C_RM_2)}{M_1 + M_2}$$

$$V_{2f} = \frac{(C_R + 1)M_1V_1 + V_2(M_2 - C_RM_1)}{M_1 + M_2}$$

Picture courtesy of Wikipedia

where

V_{1f} is the final velocity of the first object after impact

V_{2f} is the final velocity of the second object after impact

V_1 is the initial velocity of the first object before impact

V_2 is the initial velocity of the second object before impact

M_1 is the mass of the first object

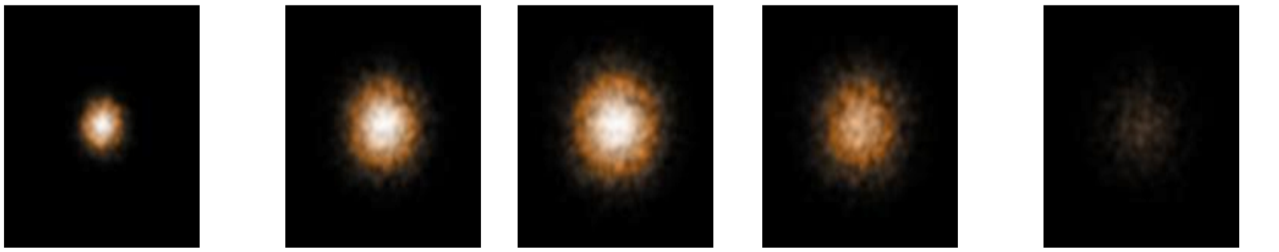
M_2 is the mass of the second object

C_R is the coefficient of restitution; if it is 1 we have an elastic collision; if it is 0 we have a perfectly inelastic collision,

Note also that equal and opposite forces are imparted on each object during the collision. Since impulse is the integral of Force over time, the force can be approximated as the change in momentum divided by time. Use this to figure out how much damage to impart on each object when a collision takes place.

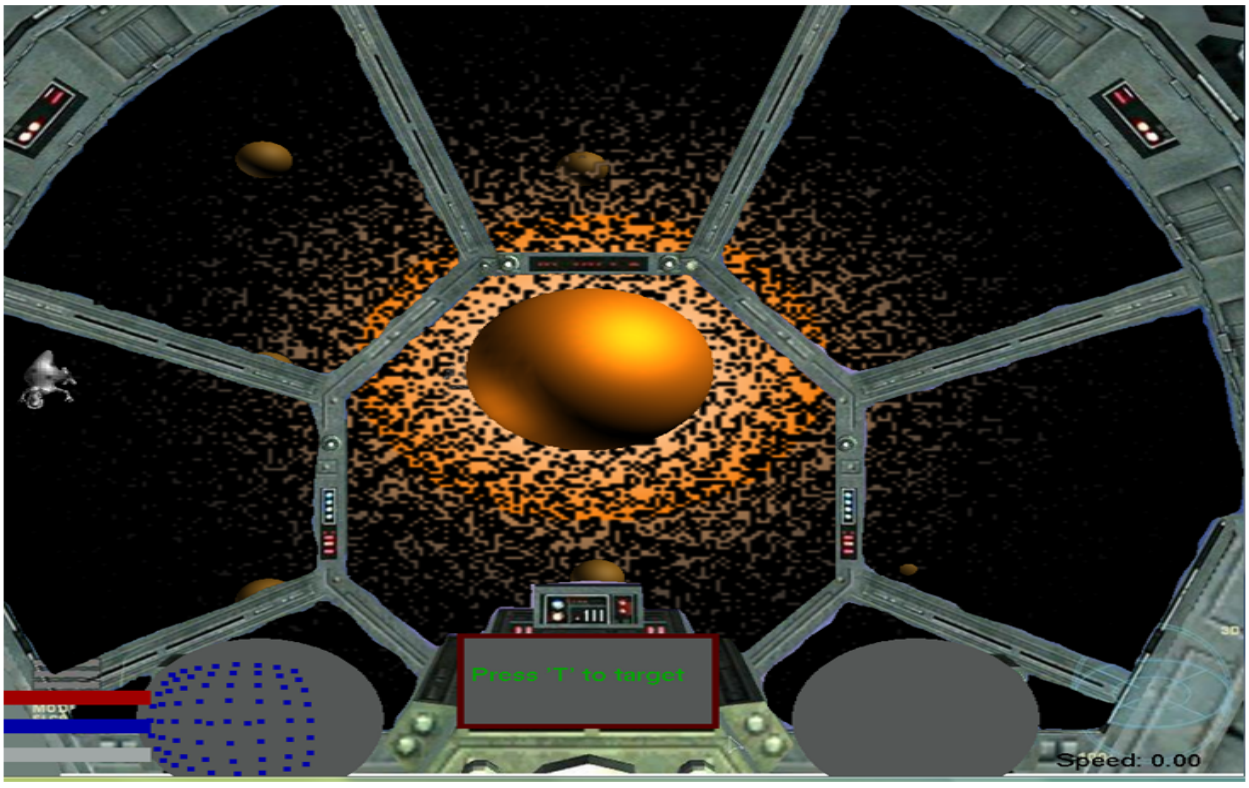
Explosions / Death

Explosions are modeled by a probability density function that evolves over time. The density function gives the probability of finding an "explosion particle" (a pixel) located some radius from the center of the explosion. We chose to make it a 2D Gaussian probability density function whose width would increase over time as the explosion grew, and then once the explosion reached its maximum width, whose height would diminish (so that the particles fade out). We modulated the color by the height of the gaussian function; so that brighter orange/white is found towards the center of the explosions and darker orange/brown is found towards the edges of the explosions.



width growing with sine | maximum | width constant, height decaying

When an object's health reaches zero, it is marked for deletion. At this point, an explosion object is created inside of it. The frames of the explosion are then calculated in real time and displayed on the screen at the center of the exploding object facing the camera (the explosions are also scaled based on how big the exploding object is). At this point, lock all control that the user or AI may have and have the ship go in a constant trajectory. Also modulate the color of the object to match the intensity of the explosion at the time. Once the explosion animation has completed after about two seconds, delete the object from the scene. If the object was the player object, switch the player into one of the ships that is still alive. If there are no more ships alive, end the game



AI

The AI are governed by a few simple rules that yield surprisingly complex behavior. Lacking the time to implement an all-out genetic algorithms approach, we innovated to find solutions that produced results on par or better than well-known algorithms (such as A* for pathfinding). The first of these innovations was the introduction of a vector field in which AI ships and obstacles radiate vectors outward from their centers along normal directions and vectors point toward the player's ship along directions opposite to its normals. Then, each timestep, an AI sums vectors from all other ships and obstacles and uses this sum to determine its orientation and acceleration. Since vectors point away from other AI ships and obstacles, an AI is repulsed by these objects. Since vectors point toward the player's ship, the AI is attracted to the player's ship. Overall, this means that the AI successfully avoids collisions with other ships and obstacles in its pursuit of the player's ship. This is all due to manipulation of the vector field; *there are no hard-coded rules for collision avoidance or player pursuit.*

A second innovation was the implementation of behavioral postures that determine how the AI thinks, literally---there are three separate "Think" functions in GameObject.cpp between which the AI seamlessly switches as it monitors its health and shield levels. The posture system is setup to operate in three modes:

- In **offensive posture**, an AI avoids collisions with obstacles and pursues the player's ship.
- In **defensive posture**, achieved when shields are below 20%, an AI avoids collisions and moves toward a "safe haven." This safe haven is dependent upon the player's position and is computed, in realtime, to be either the nearest AI ship in offensive posture or the nearest obstacle. An AI sees where the player is in relation to this safe haven and then determines a goal position that's on the other side of the obstacle from the player. This strategy effectively hides the AI from the player or, at the very least, forces the player to do much more work to attack that AI. A frustrated player might give up pursuit, giving the AI a chance to recharge its shields and return to offensive posture.
- In **kamikaze posture**, achieved when heath is below 25%, an AI avoids collisions with other AI ships and obstacles but pursues a collision with the player's ship. The rationale behind this is that colliding with the player's ship will deal more damage to the player than the AI ship could otherwise do with its laser cannon in its limited lifetime.

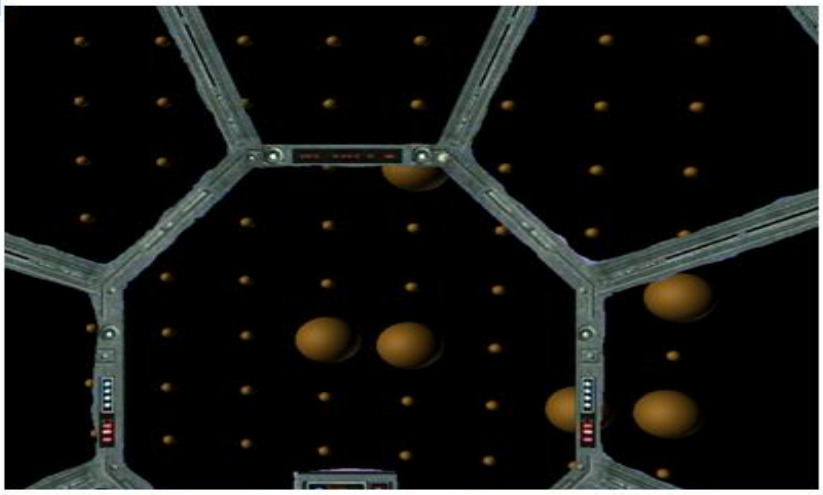
Finally, functionality was added that allows the AIs to operate as a team against the player. Each timestep, an AI determines whether it can see the player by casting a ray from its position toward the player's ship. If an intersection occurs before the ray reaches the player's ship, than either another AI ship or an obstacle blocks a direct line of sight to the player. Realizing this, an AI does the next best thing and moves toward the nearest AI who can see the player in the hope that doing so will allow it to gain a visual on the player. This is analogous to indirectly locating the player by looking at where the other AIs are shooting. This sort of advanced collective behavior would've taken a substantial amount of time to evolve using a genetic algorithms approach, if at all.

Procedural Modeling

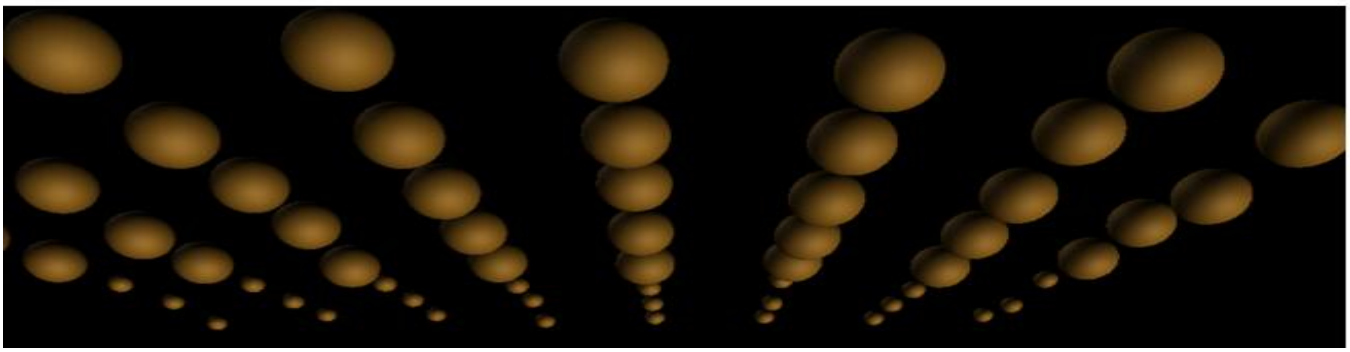
We solved the problem of developing relatively believable asteroid fields by implementing a procedural generation method that accepts a bitmap image and analyzes this file to determine where and how to place asteroids in the scene. Given a grid of pixels, this algorithm (in ProcGen () in R3Scene.cpp) uniformly samples pixels across the image and computes the luminance of each sampled pixel. If a sampled pixel's luminance is above a certain threshold, then the algorithm creates a new asteroid at a position in space that corresponds to the sampled pixel's position in the bitmap image. A large difference between the pixel's luminance and the threshold results in a very large, heavy asteroid, while a small difference produces a small, light asteroid.

It's possible to manipulate the bitmap image passed to the algorithm to grow arbitrarily-shaped asteroid fields. An all-white image with a black circle in its center, for example, translates into a field of large asteroids with a clearing in its center. We discussed improvements upon this algorithm, such as using Gaussian sampling in which the overall luminance of a neighborhood of pixels is tested against the threshold, but the current algorithm (which uses the equivalent of nearest neighbor sampling) already generates asteroid fields that are fun and challenging to traverse.

Screenshot 1



Screenshot 2



Results / Testing for Success

We met most of our goals and were able to come up with a working, playable video game. Below are some testing scenarios we used, along with an official checklist of tasks we were able to complete before the deadline...

Testing Scenarios:

- For collisions, created a file **collision.scn** with two dinopet obstacles heading directly towards each other with opposite velocities. Varied the masses and speeds of each dinopet and visually verified that the results made sense (i.e. heavier objects don't have their speed change as much, etc)
- To test procedural modeling, created two bitmaps: *test.bmp* and *test2.bmp*. To launch, type something like:
  SpaceSim collision.scn -proc_gen test.bmp
  This was also a good place to test targeting with lasers, explosions, and damage from collisions, since

there were so many asteroids to shoot down and/or collide with

- AI: To test out the defensive / kamakazie postures / targeting computer, used the file **defensive.scn**, which had two ships and two dinopets. Target the one enemy ship, and shoot at it until its shields are down to zero, and watch it go try to hide behind one of the dinopets until its shields regenerate back to 20, at which point it will go back into offensive mode again. To test kamakazie, get its health below 25, switch to external camera, and watch as the ship tries to hit into the player
- Stress Test: To test everything at once, do the following command:
  SpaceSim test.scn -proc_gen test.bmp
  This will place 17 ships in the scene, two dinopets, and an asteroid field. Watch the swarming behavior of the ships, and how the AI prevents them from getting too close to each other and colliding (in most cases). Note how they all move for the player and shoot over a certain range, but turn away to the side before they get too close and hit the player. This is a great place to test any feature or just to have fun with the game

---

Official List of Completed Tasks

- Input
  - (1) Get basic keyboard/mouse input to control roll/pitch/yaw and forward acceleration
  - (1) Get Joystick input working using the windows library

- Ship Modeling and Motion
  - (1) Create basic ship model that allows for changes with roll/pitch/yaw and velocity. Update the scene files to store specific parameters for ships
  - (1) Model all ships as cones with the top in the direction the ship is facing, a sphere over the cone to model the cockpit, and a cylinder from top to bottom (so that the user has enough info to infer orientation of enemy ships)

- Collisions / Physics
  - (1) Create a basic timestep model with position, velocity, and acceleration
  - (1) Model basic collisions treating everything as an **axis-aligned bounding box** (or a bounding sphere if an asteroid is involved). Take all pairs of objects each timestep and check them against each other (this is the best method when there are very few ships, but it doesn't scale well since it's n^2)
  - (1) Model a basic response every time a collision takes place, that takes into account conservation of linear momentum with a random energy loss
  - (2) Take angular momentum into consideration when doing the collision response
  - (3) Explosions: Have an evolving 2D sprite appear facing the camera where the explosion takes place. Make it a 2D Gaussin probability density function whose width/height change with time, and whose color is modulated by the height of the density function

- Environment
  - (1) Augment the scene files to specify environmental parameters
  - (1) Upgrade the environment to feature objects with which the player can collide, not procedurally generated. For instance, create asteroids that are spheres and that have a constant trajectory.
  - (1) Figure out how long to keep moving objects in memory; that is, if they go too far away,

delete them to prevent wasted processing effort
    - (3) Upgrade the environment to procedurally generate asteroid fields based on bitmap files

- Adversary AI
    - (1) Create barebones AI: Periodically shoot in direction of player with some random offset, keeping no closer and no farther frmo the player than x units of distance
    - (1) Upgrade AI to avoid collidable objects in the environment by keeping x distance away from such objects
    - (1) Upgrade AI to cast rays in direction of the player to determine if they have a clear shot, and to discharge their laser if they do have a clear shot and they are within range
    - (1) Upgrade AI to avoid engagement with player if its shields are below 20; have the AI hide behind the nearest obstacle
    - (1) Upgrade AI to collide with the player if near death (kamikaze mode)

- Game-Specific / Other:
    - (1) Create a global camera based on the bouding box of all ships/obstacles where they start. Toggle this camera with "O"
    - (1) Create a 3D camera that tracks the player's ship from the outside. Toggle with "S"
    - (1) Allow the player to cycle through all of the available ships in the scene by hitting "C"
    - (1) Upgrade player and adversaries to have health, shields, and a rate of repair
    - (1) Create a basic weapons system (lasers), and do ray tracing to determine if they hit any objects in the environment.
    - (1) Create a cockpit overlay for the user's ship viewpoint that has transparent windows
    - (1) Create a radar system that shows users what's in front of and behind them
    - (1) Create a targeting system that allows users to view information about adversaries (health, distance away, etc.); toggle with "T"
    - (1) Add some basic sounds for collisions, laser fire, and explosions (no 3D sound; they are they same amplitude for any distance / direction)

Discussion / Conclusion

We are extremely satisfied with the progress we've made so far. In just a few weeks, were were able to develop a fully functional 3D space flight simulator from the ground up. Although it's not the most efficient it can be yet and the artwork is primitive (we chose to focus on a lot of the gameplay issues from the onset), it demonstrates the concept very well and all basic features are there. We knew we had something good when it was already fun to play at this point. For this reason, it makes sense to continue development on the project and to refine it, because it could be a hit video game with a few improvements! First of all, we will need to implement some spatial partitioning algorithms to speed up collision detection and AI operations; because right now in collision, every object is checked against every other object (n^2), and for each AI ship, all objects and ships are tested to help construct the vector field, as well as doing ray tracing against every object when lasers are fired (or just to determine if the player's ship isn't blocked). Some sort of dynamic spatial partitioning would greatly speed this up and make it possible to add a more or less arbitrary number of AI and obstacles without significantly decreasing the framerate.

Another thing we'll need to do soon after that is to create more complicated mesh models for ships and asteroids. To accomodate the new mesh structures in the collision model, we also should do collision tests beyond bounding boxes and bounding spheres (we can just use them as trivial rejects). Then, we should make

different types of ships with different speeds, maneuverabilities (the code is there right now for this...just change the max centripetal accelerations), levels of aggression, etc.

This project gave us an opportunity to explore some topics we did not go over in class, such as procedural modeling, explosions, and AI, all in the context of a fun, interactive application. We also got a chance to try out some of the basics of particle systems at an interactive level, which was something we never did an assignment on.

References

- http://www.gamedev.net/reference/articles/article1679.asp
- http://basic4gl.wikispaces.com/2D+Drawing+in+OpenGL
- http://www.opengl.org/resources/faq/technical/transformations.htm
- http://www.gamasutra.com/features/20000203/lander_01.htm
- http://www.phy.ntnu.edu.tw/ntnujava/index.php?topic=4
- http://en.wikipedia.org/wiki/Inelastic_collision
- http://www.opengl.org/documentation/specs/glut/spec3/node75.html#SECTION00011000000000000000
- http://www.opengl.org/documentation/specs/glut/spec3/node64.html#SECTION00081900000000000000
- http://msdn.microsoft.com/en-us/library/ms709354(VS.85).aspx