Simulating a Starship Battle
6.837
Fall 2002

Authors
Team 2
Christopher Taylor, Jonathan Wolk, Reuben "Benji" Sterling

December 6, 2002
Abstract

Design and implementation of an engine to simulate and render a space battle between fleets of starships. All ships are AI controlled, so a battle can end without any user input. The user is given small amounts of camera control as well as pausing and time related control. The program is intended to render the battle in real time. The style of fight could be reminiscent of Star Wars. A number of weapon types were to be supported, depending on time constraints. Ideas for these included laser bolts, homing missiles, beam weapons, and flak guns.
Contents

1 Introduction ................................................. 1

2 Goals ..................................................... 1

3 Achievements ............................................. 1
  3.1 Artificial Intelligence .................................. 1
  3.2 Physics .................................................. 2
  3.3 Collision Detection ..................................... 3
    3.3.1 Optimizations ....................................... 3
    3.3.2 Physical Collisions and Updated Movement ............. 3
    3.3.3 Future Collision Ideas ................................ 4
  3.4 Camera Management ..................................... 4
  3.5 Rendering ............................................... 4
    3.5.1 Ship Models ......................................... 4
    3.5.2 Shield Effects ........................................ 5
    3.5.3 Explosions ............................................ 5
    3.5.4 Particle Trails and Engine Glow ......................... 6
    3.5.5 Skyboxing ............................................. 6
    3.5.6 Weapons ............................................... 6
    3.5.7 User Interface ........................................ 7
    3.5.8 Frustration Culling ................................... 7

4 Lessons Learned ............................................ 8

5 Acknowledgements ......................................... 9

6 Appendix .................................................. 10

A Compiling the Program .................................... 10

B Running the Program ...................................... 10

C User Commands ............................................ 10

D System Requirements ...................................... 10

E Ship/Faction Description File Format ...................... 12
List of Tables

1  Default User Commands .................................................. 11

List of Figures

1  Shield and Explosion Effects ............................................. 5
2  Particle Trails and Engine Glow Effect .............................. 7
1 Introduction

Our objective was to create a good-looking engine using Direct3D that reflects a "believable" space battle, insofar as that is possible. Doing so meant designing spectacular effects, doing realistic physics and collision calculations, implementing credible AI, and doing so under the constraint that all calculations and rendering be done in real time. Thus, we weren’t particularly interested in a single computer graphics technique so much as creating an effective combination of as many as we needed. To some extent we were motivated by the desire to produce something reminiscent of the more impressive computer games we have played.

2 Goals

Our goals for this project were rather broad. We wanted the focus to be on smaller ships - fighters and that sort of thing. We hoped to add larger ships, on a scale of 10-20 times larger than the fighters. We planned to include interesting graphical effects such as particle trails, explosions, and weapons. Another area of interest was camera management - making it as easy and intuitive as possible for the user to watch the battle and pay attention to ships/events of interest. We also took some naive guesses at the sorts of optimizations that would be required: level of detail scaling, view frustum culling, data structures for spatial coherency, etc. None of the group members had attempted a coding project of this type or scale before, so we had little prior knowledge of what would work well and what would not. We were interested in experimenting with advanced capabilities of modern graphics hardware, particularly programmable shaders.

3 Achievements

We think we essentially succeeded at the task we set for ourselves. Given another week of development time, we think even the most ambitious of our goals could have been fulfilled. It would be excessive to say that our engine is perfectly optimized, but it does what we set out to do, and could be made to do more with relatively little difficulty. We did not, in the end, successfully include capital ships in the simulation. Some of the code was written, but we didn’t have enough time to bring it to the level of the rest of the project. As a result, we decided to leave it out entirely. There was insufficient time to add sounds using DirectSound.

The following aspects of our project are particularly notable:

3.1 Artificial Intelligence

Artificial intelligence proved to be the least accessible problem for our group to solve. Especially since the rendering and physics were under development from the day we started to the day we
stopped working entirely, making meaningful progress was difficult. If we continued to work, this is one of the areas were improvement could be made.

In the current system, each ship independently chooses a target at random from the ships in all opposing factions. It attacks until that ship is dead, and then chooses a new target until there are none left.

The actual attack code we went with is simple but effective. The ship lines up with its enemy, and fires when it is close to a straight line. It accelerates forward any time it is facing its target. If it collides at any point, or gets too close (with 10 units) of its enemy, it chooses a point a good distance away and flies there before making another attack run.

The resulting behavior looks intelligent most of the time. Occasionally there are friendly fire accidents, and collisions between ships are decently frequent, especially when a good number of ships get into a melee. However, all battles we ran ended with a clear winner - all ships on opposing factions were destroyed.

All of the AI code ran in a separate thread from the main UI and rendering thread. The AI thread simply looped over a vector of AIOBJECTs and called their AI function. All ships implemented the AIOBJECT abstract class.

3.2 Physics

We implemented a complete linear physics system (acceleration, velocity, displacement), and rotational physics (velocity, displacement only). The movement system was set up using a discrete time step system where the concept of “time” was divided into smaller time steps. During these time steps, accelerations and velocities (both linear and rotational) are assumed to be constant to simplify the math. If these time steps are sufficiently small (usually around one hundredths of seconds) then these assumptions are not far from reality. Within each time step, velocities are updated and then the objects’ transforms are moved and rotated accordingly.

The rotational movement was solely given velocities (no accelerations) for a few reasons.

First, examination of other flight sims suggests that this decision is not uncommon. Secondly, this made for the AI to be much better. It is very hard to get a ship to rotate so that it is facing a certain way if you have to impulse and accelerate then decelerate at an exact time. And lastly, it made the math much simpler. All rotations and rotational math uses the Quaternion format so that installed DirectX math libraries could easily be used. One thing that was a pain with the Quaternion movement came with corner cases such as π and 2π because in Quaternion form, a velocity of 4π may look the same as 2π, but they are not. A little bit of trickery bypassed this problem though.

Just like the AI system, physics runs in its own thread. This thread processes PhysicalObjects in order, moving them using an Euler approximation based on the time since last update. The physics thread also does collision computations and uses a callback in the PhysicsTask class to do weapons updates.
While the overall concept of the Physics system is rather simple, the overwhelming amount of math involved to do simple calculations made making this system tedious and very time consuming. It also took a lot of energy to find ways in which to make the system faster (such as some ways mentioned in I-Collide).

The final physics system yields movement with no glaring discontinuities. It was easy to integrate with the renderer and the AI, and it does not hog CPU time. We consider it a success.

3.3 Collision Detection

The collision detection problem we faced was complicated. In some cases, over 100 objects needed collision testing every time a physics update occurred. These would be roughly ten percent "large" spherical objects (i.e. fighters), and the remainder would be laser bolts and missiles (modeled as points). We were interested in detecting all collisions except those between points.

3.3.1 Optimizations

We opted for a system roughly modeled on I-Collide, a system developed at the University of North Carolina, Chapel Hill for detecting sphere/sphere collisions [2]. The I-Collide system uses an axis-aligned bounding cube for each scene object and exploits spatial coherence to quickly cull pairs that will not collide in a given time step. Then the remaining pairs can be passed to a more exact collision routine (in our case moving sphere/sphere collisions). This system allowed incremental collision testing to occur in expected $O(n)$ with the number of fighters.

For point/sphere collisions, we decided against heavy optimization. The collision could be modeled as a line segment/sphere intersection problem, and we felt that this was sufficiently fast [1]. All points are checked against all fighters, giving us $O(mn)$ performance, with $m$ points and $n$ spheres. In actual practice, this choice seemed valid, since $n$ is fairly small compared to $m$.

3.3.2 Physical Collisions and Updated Movement

When a collision is detected between two objects, the objects are updated to their point of collision. Their resultant velocities after the collision (all collisions are assumed to be perfectly elastic) are calculated using conservation of momentum principles and then, using these new velocities, the objects’ positions are updated to the end of the time step. This last little bit may cause some strange bugs if the objects are moving fast enough because we blindly advance them to the end of the time step which may cause the ships to “pass through” another ship. This does not happen at all since the ships have air resistance to slow them down and because the models are so large, the ships will never pass through one another, but may catch a collision a time step late.
Collisions are processed using callbacks to the PhysicalObjects involved in collision. Weapon hits are processed using a separate callback in the weapons system.

3.3.3 Future Collision Ideas

We planned to implement an axis-aligned bounding box tree system for stationary capital ships (such as space stations), and were close to finishing, but were forced to abandon the work due to lack of time.

3.4 Camera Management

The camera management is perhaps one of the most impressive aspects of the project. We give the user the ability to translate, rotate, and zoom using key commands. Double clicking on ships (or their portraits in the right user interface panel) causes the camera to fly to the ship. Holding down the control key during the double click causes the user to tether to the ship. While tethered, the user can still move normally, but his transform is with respect to the center of the ship rather than with respect to the world space origin. The simulation can be paused at will, and the camera can still be moved at will during the pause. Similarly, the simulation can be forced to run at 1/4, 1/2, or double speed. In particular, this ability to manipulate time and camera position enables the locating and capturing of some fairly dramatic screenshots. While the rendering, physics, and AI systems are more difficult, and more essential, we think it is the camera that really makes the program so compelling.

3.5 Rendering

Rendering is by far the most important aspect of our project. Since this is a Graphics based class, our main objective is to display cool visual effects and movement. Rendering is in charge of displaying images on the screen with the help of DirectX and hardware acceleration abd is therefore the heart of our project.

3.5.1 Ship Models

The ship models themselves were downloaded from the internet in varying file formats, and converted using an assortment of file format converters to the .x format. (See acknowledgements) Then our system takes over, automatically generating collision bounds, loading the correct textures and materials, and rendering the models to the screen. We also attach appropriate particle streams and weapon emplacements. In the end, creating our own models proved quite difficult and time consuming, and we felt that 3d modeling was somewhat peripheral to the core of our project.
3.5.2 Shield Effects

The shield effect was our earliest attempt at special effects, and was often retooled. The initial stab featured a sphere per fighter. Hits on the shield were represented by the normal at the point of impact, and points on the shield were shaded by a function involving the dot product of the point normal with each of the hit normals [3]. Calculation on the CPU proved exorbitantly expensive, so the dynamic shield incurred unacceptable speed penalties. We then tried a textured approach. This approach proved fast, but reasonably ugly. Later in the development process, the shield code was rewritten a third time using a programmable vertex shader instead of the CPU to calculate shield colors. This method brought down the load on both the CPU and the AGP bus to a sufficient degree that this method proved feasible. This experience, along with the explosion effect, was a dramatic indication of the power of programmable shaders.

Such shields can be seen in Figure 1.

![Figure 1: Shield and Explosion Effects](image)

3.5.3 Explosions

The explosion effect is easily the coolest thing we did. The system maintains two vectors of Explosion objects, each of which contains its own vertex and index buffer. One vector is active,
the other is inactive, and Explosion objects are reused to reduce dynamic memory allocation calls. When an explosion is triggered, the vertex buffer is filled with randomly generated quads in a special vertex format that includes information such as initial velocity and start time. There's also an extensive array of constants to control color and speed modulation of each of the five texture types over the course of the explosion. The world transform is set such that the quads are all drawn parallel to the screen and in the correct world space location. Thereafter, all physics and color calculations are done in a vertex shader on the GPU. This approach allows the vertices to stay on the graphics card for the duration of the effect, thus minimizing the load on the CPU. Each of the textured quads in alpha blended into the scene. More explosions can be seen in Figure 1.

### 3.5.4 Particle Trails and Engine Glow

The particle system is used for everything from wingtip trails to engine glows to the tracking missile effect. Given an initial displacement, an initial velocity, a transformation, a rate of generation, and two angles to represent the dispersion angles, the ParticleEmitter class generates a stream of textured point sprites meeting the parameters. All point sprites throughout the scene are drawn in the same rendering pass to minimize changes of render state. The particles are alpha blended into the scene. The results give a reasonable impression of engine glow, and depending on the colors chosen, can create rather interesting effects like the tails on the tracking missiles. The performance of the system is good in practice, since the total vertex count of the system is lower than the vertex counts on some of our models. 30-40 points is enough to handle a single particle emitter when they are cycled effectively, and the rendering of a few hundred point sprites is negligible on the sorts of graphics card that have programmable vertex shaders. These effects are seen in Figure 2.

### 3.5.5 Skyboxing

The skybox is a simple idea that turned out well. Simply draw a textured cube centered on the eye point, with z-buffering disabled before the rendering begins. The textures were generated with Starbits 7. The sun and planet were later added using Photoshop. The background adds nice ambiance without being too complicated or slow to render.

### 3.5.6 Weapons

The weapons system ended up being sufficiently flexible for our needs. The system contains an inactive list and an active list, which operate similarly to the ones in the explosion system. The processing of the bolts/missiles happens in the physics thread via the PhysicsTask class. The tracking missiles inherit from the TransformWrapper class, so that they can receive particle trails.
The actual weapon drawing is done with point sprites, and all bolts are rendered in one pass, so, like the particle system code, the rendering happens quickly.

3.5.7 User Interface

The user interface is a fairly simple idea - it consists of textured quads rendered with an orthographic projection and alpha blending after the main rendering has completed. The brackets and faction tags are also drawn during the user interface pass, and are simply alpha blended quads centered at the ship location. For the bracket, an additional translation and scaling factor are added to ensure that the bracket stays above a minimum size and remains inside the far clipping plane.

3.5.8 Frustum Culling

We implemented a sphere-based frustum culling system for fighters. The idea is generic, but the speed benefits are good. Especially after the ships start to spread out, frustum culling allows significantly more ships to be in the engine. Each fighter rendering can require up to 3000 vertices to pass through the graphics card.

Individual contributions stayed along the lines of what was described in the project proposal.
Most of the stunning visual effects, the heart of the project, were a result of Chris spending many late nights playing around with the different features of his GeForce 4 graphics card. Jon spent a good deal of time implementing the system’s physics and collision detection and handling algorithms. Finally, Benji spent a lot of time implementing an extensive framework for the system’s AI, which, unfortunately, was scraped at the end for a much simpler implementation. In addition, he implemented a file reading system so that ship and faction information could be described in a text file instead of being directly defined in the code.

4 Lessons Learned

We learned a great deal about the use of DirectX and the design of a rendering engine. Given the chance to go back and write the project again, there are a number of things we could do better. For instance, it turned out that spacial coherence between ships is relatively unimportant in terms of rendering, and we would have been better off conserving render states and rendering all fighter bodies in one pass, and shields in a second one. However, by the time we realized that, it was too late to go back and fix it. We learned how to use texture blending operations and alpha blending to create a variety of effects. When we started the project, we didn’t know much about Direct3D, and certainly nothing about how to generate the effects we were looking for. By the end, we were getting good at visualizing the effect and seeing how we could decompose it into the operations we’d need to do to render it optimally. We learned a bit about how to identify places to use vertex shaders, and the vertex shader assembly language. The progress we made was exponential as we got more fluent with the Direct3D API. We learned a lot about transforms, and chaining them together. We used quaternions for part of our rotational physics, and discovered the hard way what their limitations are. As part of designing the collision detection system, we gained passing familiarity with a number of ways to do collision detection in realtime. We also had to gain passable skills with Photoshop as a tool for making textures. We learned to use the debugging features of Visual C++ 6.0, which are amazingly useful when used correctly. Very often a half dozen clicks would lead one straight to the source of the problem.

DirectX was idiosyncratic, but once we figured out the gist of it, it was generally easy to use. It was hard to debug in a lot of cases though, because very often, the only symptom would be that nothing showed up on the screen. Video resource leaking was also very difficult to debug, because the debug information the libraries generated was minimal at best, and mostly consisted of illegible hex codes. The DirectX framework for Visual C++ was very good. It got us off the ground quickly, and was a solid foundation to build on even when we started to understand what we were doing.

The modeling formats were difficult to work with though. The D3DX libraries provide loaders for the .x file format. The DirectX Utilities include a program conv3ds.exe, which converts 3D Studio Max files into .x files. However, it worked at best 25 percent of the time
on .3ds files we downloaded. It never worked, to our knowledge, on Lightwave files that were exported as .3ds. That aspect of the project was a continual challenge. Everyone models in a different modeler, and in many cases, the formats are wildly incompatible with no reliably correct conversion programs.

Photoshop, like DirectX, got easier to use the more often we used it. Drawing our first half dozen lines took an hour, but towards the end we could make cool things fairly quickly. We provide the following artifacts as part of our project:

Executables, Source code, models, images, and Visual Studio project/workspace files as zip

benster.mit.edu/6.837/team02.zip

Oral presentation materials

benster.mit.edu/6.837/Graphics Final Project/Model Viewer/oral presentation.ppt

This report

benster.mit.edu/6.837/team02report.pdf

Screenshot library

benster.mit.edu/6.837/Graphics Final Project/Model Viewer/screenshots/

5 Acknowledgements

In addition to the items listed in our bibliography, we used the following:

Microsoft DirectX SDK. We used the Direct3D and DirectInput portions. Downloadable from www.microsoft.com. The SDK documentation is mostly good, but is sparse enough that sometimes it can be difficult to see the intended use of many functions. Sometimes, the documentation is poor enough that the intended purpose of the function is unclear.

Microsoft DirectX Framework. Provides window creation and DirectX initialization code, as well as some helper routines. Included with the DirectX SDK.

nVidia Effects Browser and code samples. Provides quite a few sample vertex programs whose code is free to look at or reuse. The explosion images we use in our project come from the Effects Browser. All code is our own, however.

www.gerf.org/ nixon. This is the source of the Assault Gunboat model and textures.

Darksaber’s Opt Station (http://www.darksaber.freeserve.co.uk/). This is the source of the other TIE fighter models. It was also the source of the .opt to .3ds converter we used.
www.turbosquid.com. This is the source of all four Greenie models. They are provided there as a free download.

Jennifer Chen, a friend from Harvard, helped us with some of our Photoshop work, particularly in the final formulation of the skybox graphics.

http://members.xoom.com/echomag. The home page for Starbits 7.03, a program for creating starfields and galaxy images. We used this extensively for our skybox images. The web page appears to be down recently, but the documentation with the program suggests that it is also available on www.download.com

6 Appendix

A Compiling the Program

Open 'Graphics Final Project.dsw'. You will have to set up the DirectX 8.1 SDK on the computer, and configure Visual C++ as stated in the DirectX 8.1 documentation. If this is done correctly, the program will compile and run out of the box.

B Running the Program

Running the program directly produces a default scene. The name of a scene file can be passed as a command line argument. If this is done, the program will open using this initial scene.

C User Commands

Double clicking on a ship portrait in the selection panel is equivalent to double clicking on the ship itself.

You can use the mouse to access the faction menus in the lower left corner. Clicking on a faction name brings out a list of the ship types that faction has surviving. Clicking on a ship type produces a list of the ship names. Clicking on a ship name selects the ship.

D System Requirements

The target system should sport a recent processor (1+ GHz), a reasonable amount of RAM (probably 256 MB). It needs 10 MB of hard drive space for program files and images. The graphics card must have programmable vertex shaders and point sprite capabilities (Geforce2, Geforce4 Ti). Less powerful systems cannot display most of our effects, and are thus not advisable.
## Keyboard Commands

<table>
<thead>
<tr>
<th>Key(s)</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>W,S</td>
<td>Translate forward, backward along the XZ plane</td>
</tr>
<tr>
<td>A,D</td>
<td>Translate left, right along XZ plane</td>
</tr>
<tr>
<td>Q,E</td>
<td>Translate up, down along the Y axis</td>
</tr>
<tr>
<td>Up,Down</td>
<td>Pitch up, down</td>
</tr>
<tr>
<td>Left,Right</td>
<td>Yaw left, right</td>
</tr>
<tr>
<td>+,-</td>
<td>Zoom in, out</td>
</tr>
<tr>
<td>U</td>
<td>Untether, if tethered</td>
</tr>
<tr>
<td>T</td>
<td>Activate/Deactivate faction tags</td>
</tr>
<tr>
<td>X</td>
<td>Deselect selected ship, if any</td>
</tr>
<tr>
<td>F2</td>
<td>Configure input</td>
</tr>
<tr>
<td>F3</td>
<td>Configure display (full screen not fully supported)</td>
</tr>
<tr>
<td>F4</td>
<td>Toggle frame rate statistics</td>
</tr>
<tr>
<td>,</td>
<td>Slow down time by half</td>
</tr>
<tr>
<td>.</td>
<td>Speed up time by 2x</td>
</tr>
<tr>
<td>P</td>
<td>Toggle pausing of simulation</td>
</tr>
</tbody>
</table>

## Mouse Commands

<table>
<thead>
<tr>
<th>Button</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single click</td>
<td>Select ship</td>
</tr>
<tr>
<td>Double click</td>
<td>Zoom to ship location</td>
</tr>
<tr>
<td>Ctrl-Dbl</td>
<td>Tether to ship</td>
</tr>
</tbody>
</table>

Table 1: Default User Commands
Our two primary test systems were:

Pentium 4 2.53 GHz
512 MB PC1066 RDRAM
Geforce4 Ti 4600

Pentium 4 2.0 GHz
256 MB RAM
Geforce3 Ti 200

E  Ship/Faction Description File Format

The file format for describing factions and ships is given below. Characters and words to be copied verbatim are listed in boldface. User defined values are denoted by angle brackets. Line breaks are extra spaces for readability only. They are ignored by the parser. Comments are denoted by “//” are not part of the file format.

Factions:
//you may define as many factions here as you’d like //but all definitions must lie between the two bold “Factions” delimiters
“< factionname >” //quotes are required
< float > < float > < float > < float > //ship color (rgb alpha)
< float > < float > < float > < float > //user interface color
< float > < float > < float > < float > //shield color
< float > < float > < float > < float > //engine color
< float > < float > < float > < float > //wing tip trail color
< float > < float > < float > < float > //selection cursor color
< float > < float > < float > < float > //text color

“Factions:” //quotes are required
< int > //faction number as define above
< float > < float > < float > // (x,y,z) of ship’s linear displacement
< float > < float > < float > // (x,y,z) of ship’s rotation axis
< float > //angle of ship’s rotation around the defined axis
References

