Core13

Seth Black

CPE 476/CPE 492

Advisor: Zoe Wood

Spring 2011

Introduction

Core 13 is an arcade styled mech battle game. Unlike other mech games, rather than emphasize realism with a slow and clunky mech, the player is provided a fast paced battle the leaves no time to idle around. As fun as it sounds to pilot a giant robot tearing down its surroundings, such games fall short when the players' patience is tested. At heart, such games are just first person shooters which require a fast paced game play. On a more technical side, the Core 13 project is as much a game as it is a technique for making a game. The project includes a level editor and model builder built from scratch. Everything down to the texture was built from scratch with audio being the only exception. This project was written in C++ with the OpenGL API and SDL library, allowing the game to be built for Mac, Windows, and Linux.

Description

Core13 is a mech-themed first person shooter. Very light on story, the plot and motivation of the player can be summed up in the following bullet points:

- You have a remote controlled mech loaded with an assortment of missiles.
- There are a number of automated robots loaded with missiles and programmed to attack your mech.
- Blow them up before they blow you up.

Movement of the mech is handled with the WASD keys for horizontal and the space bar for vertical thrusters. The mouse is used to control where the mech is looking.

The main game mechanic of Core13 is the cores. Twelve power cores control the mech's mobility and represent the "health" of the mech. The player can toggle the activity of any of the twelve cores by pressing the number keys, '-', or '='. The more cores that are active, the faster the mech moves, and the more powerful the thrusters are. Too few active cores and the mech can no longer lift off the ground; all cores deactivated and the mech can't move or fire at all. The trade-off for mobility is that all active cores are damaged when the mech is hit, and the damage per core is constant regardless of how many cores are active. When any core is fully damaged, the core fails catastrophically, destroying the mech. If one of the other cores is inactive when an active core becomes critical, the damaged core will immediately deactivate while the inactive core takes its place. Because core replacement happens before core failure, fewer active cores equates to a larger buffer before destruction.

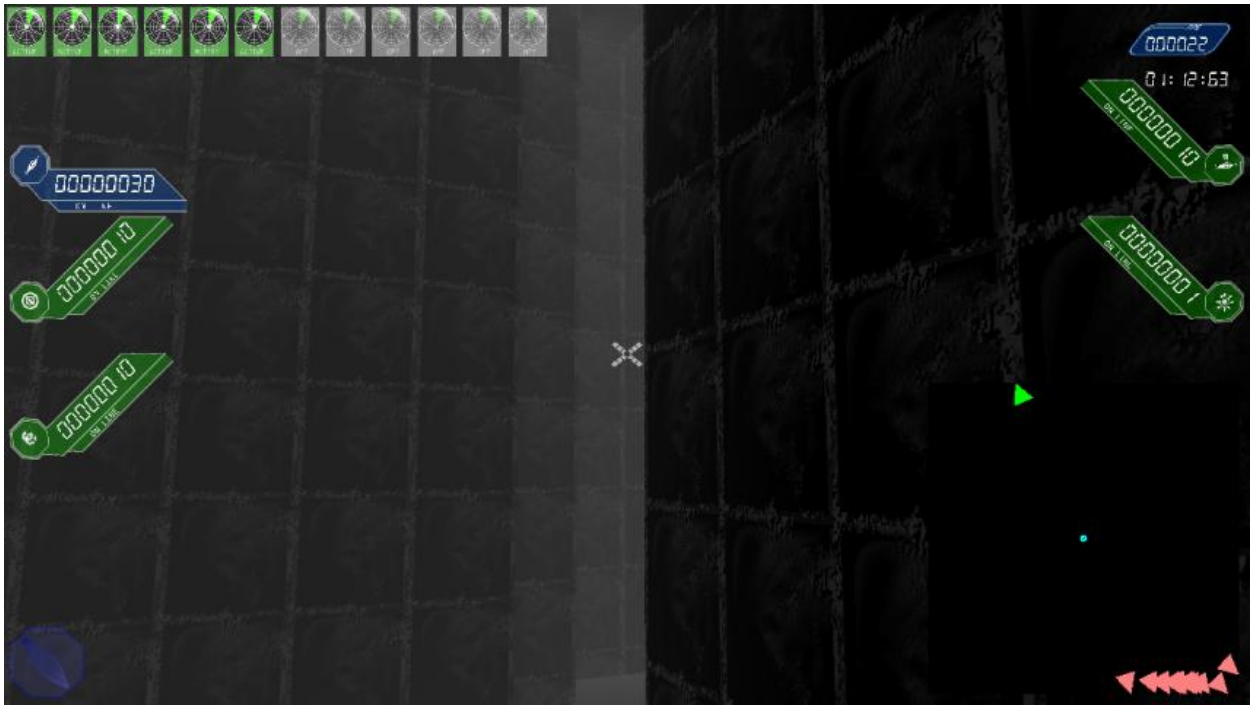

Figure 1: Six Active and Six Deactivated Cores

The player has six modes of attack: one primary (fired with the left mouse button) and five secondary (fired with the right, selected with the scroll wheel). The primary attack fires a missile that goes straight until it impacts or times out; six of the same missile are fired with the first secondary attack. The next secondary is the homing missile. If the player has a target locked-on, the missile will home in on the target as long as it has line of sight, and will curve to avoid colliding with buildings. The third is the drunken missile, which randomly swerves in flight. The fourth is the seeker missile, which homes much like the homing missile. Unlike the homing, the seeker doesn't track the target, but instead goes after the nearest enemy in a cone in front of it. The last is the hydra missile, which goes straight for a distance before splitting into ten seeker missiles.

The player is also provided with radar in the lower right corner of the screen. Buildings are shown in white or grey and enemies in red, the shade varying according to the height relative to the player. Enemies outside the range of the radar are marked with a red arrow along the edge. The view rotates with the player such that forward is up; North is represented by a green arrow.

Core13 features three enemy types. The turret, the only immobile enemy, fires ordinary missiles at the player when it has line of sight. The hover, a small Frisbee-ish robot, flies around the map searching for the player. When it finds the player, it fires drunken missiles until destroyed or it loses the player. Because they lack accuracy, the hovers are only really a threat in numbers; a swarm of hovers can bring down the player rather quickly. The final enemy type is the mech. The mech will follow the player as best it can (the mech lacks the flight ability of

the player) and fires ordinary missiles when it has line of sight. If the mech is prevented from moving to a location where it can fire on the player, it launches a pair of position homing missiles that will arch to hit the player if they don't move quickly. The mech is the largest and hardiest enemy of the set, capable of taking numerous salvos before exploding.

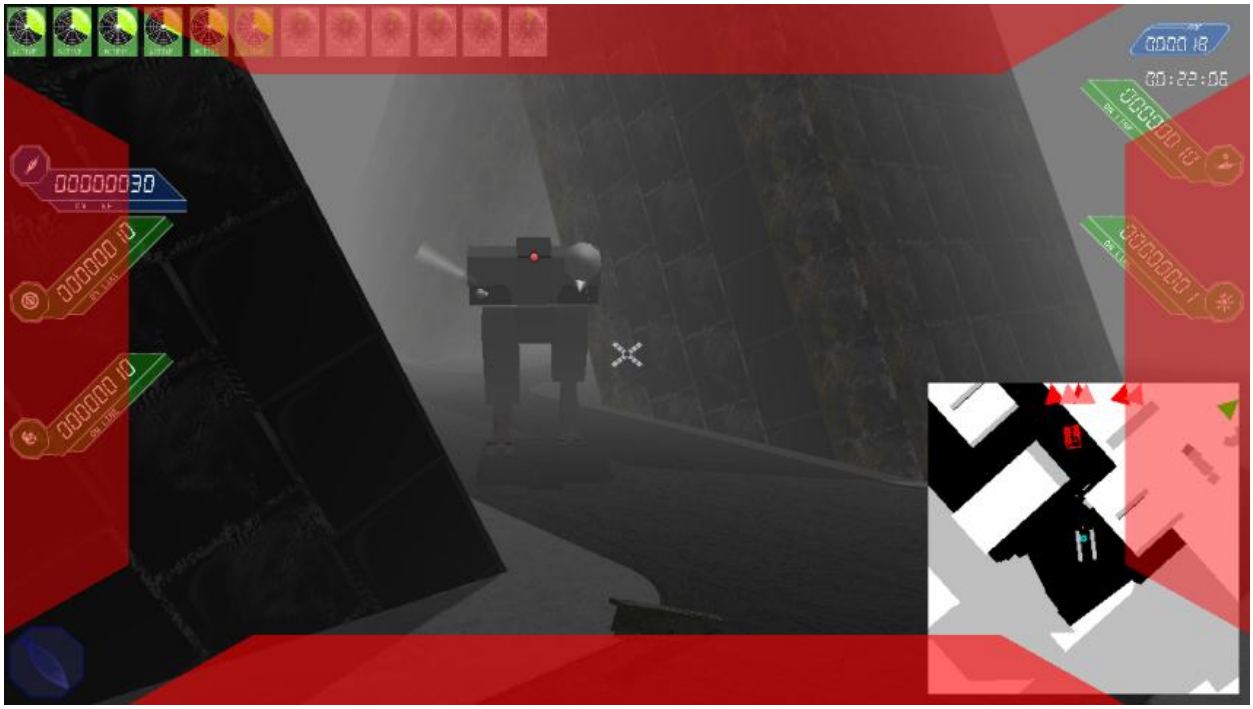

Figure 2: The Mech

Core13 has a map loader, allowing players to choose which map they want to play in. Overall, the maps follow a cityscape theme, though with varying levels of decay. All maps are enclosed by a green energy barrier that serves to keep the destruction from spilling out.

Technologies

- Animated models: Joshua Maass
- Audio: Jereis Zaatri
- Pixel shaders: Seth Black
- Spatial data structures: Seth Black
- Collision detection: Seth Black
- View frustum culling: Jereis Zaatri, Seth Black
- Particles: Seth Black
- Shadows: Seth Black

Collision Detection

An important mechanic of Core13, one of the first to be coded, is collision detection: determining if, and occasionally when, two objects are in contact. The reason this is so important is that it forms the basis of the game play: Core13 is a first person shooter, and collision detection is necessary for determining if a shot hits or misses.

It is hard to define an algorithm for collision detection. On the one hand, there is no real generic algorithm for testing collision between to objects. On the other hand, there are specific algorithms for the collision of any combination of shapes. An algorithm for two points colliding, two axis-aligned boxes colliding, two unaligned boxes colliding, a point and a box colliding, etc. Ultimately, the generic "algorithm" for collision detection is: determine what the shapes are that are being tested, then run the relevant algorithm for that pair.

While there is only this one algorithm to use, there are two ways it can be implemented, with different advantages and disadvantages to each. Because of the various circumstances for collisions in Core13, the project uses both implementations.

In the discrete implementation, objects are moved and then collision is tested. Time is discarded from the equation, all objects are effectively static. The benefit of this method is simplicity. A 1x4x9 box remains a 1x4x9 box, whether it's resting on the Earth or orbiting around Jupiter. Because the box doesn't change shape for the test, no additional algorithms are necessary. The downsides of the method are that, because time is left out, the collision tests can't determine anything that is dependent on time, such as the precise location of impact at the moment the objects collided, nor can they detect any collisions that happened between the

current and previous time slice. As a result, this implementation is best used for scenarios where precision is unneeded and collisions are likely to last longer than the intervening slices. In Core13, the tests to ensure that the player and enemies don't walk through terrain use the discrete method of collision detection.

In the continuous implementation, the trajectory of the objects are calculated and tested for collision before their positions are updated. The advantage of this method is the disadvantage of the other, and vice versa. Continuous collision detection can determine the precise location of impact and won't miss collisions that happen too quickly. Unfortunately, this comes at the price of more complicated shapes and algorithms. A sphere turns into a cylinder with hemispherical caps, for example. As such, continuous detection is best used only when needed, and the shapes involved are best kept simple. In Core13, the projectile collision and line-of-sight tests are handled using the continuous method.

Figure 3: Discrete vs. Continuous Collision Detection

Another important part of collision detection is the shapes. Improving collision detection generally involves improving the accuracy of the bounding bodies, the sets of shapes that define each object. While it is possible to define any given object as a set of cubes, and improve accuracy by shrinking the cubes, far greater efficiency and accuracy can typically be

achieved by increasing the variety of shapes to choose from. The drawback is that, at worst case, each new shape requires an algorithm for each existing shape, and the number of algorithms grows quadratically.

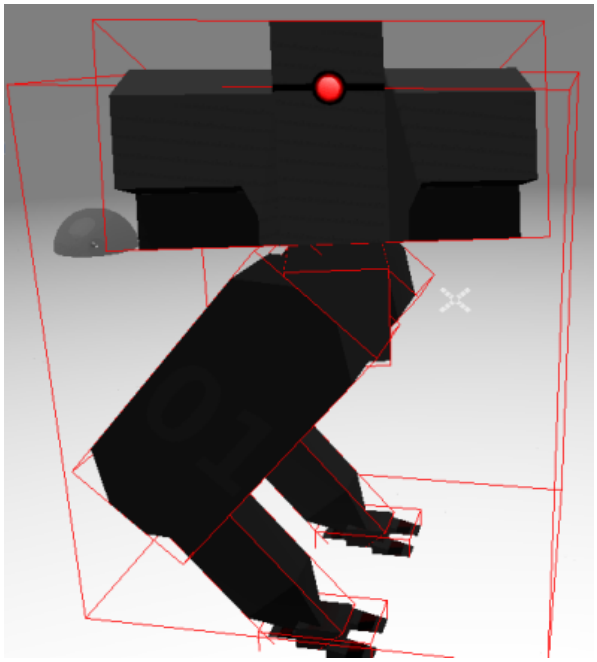

Figure 4: Mech with Bounding Boxes Shown

Spatial Indices

The biggest issue that arises from collision detection is performance costs. Complicated bounding bodies take time to test, and even if the bodies are very simple, checking all possible collisions can take time if there are numerous objects. The solution to this problem is cut out superfluous tests and focus on only the situations where the two objects might actually collide. One way of doing this is with spatial indices, which subdivide the world into smaller spaces.

The defining characteristic of a spatial index is how the space is divided. In the octtree, the world space is divided in half along each of the three dimensions, forming eight sections of equal size; these sections are then similarly divided an arbitrary number of times or until the given section is empty. In the *k*d-tree (*k* dimensional), the world is divided in two along alternating dimensions, with the dividing plane being chosen so the two sections contain a roughly equal number of objects. These and other methods have the benefits of smaller memory costs and quicker search times. The problem with them, which is what led to choosing the following algorithm, is the code complexity and the performance overhead to update and rebalance the trees when the player or enemies moved.

Instead, the algorithm chosen for Core13 is the grid-based index. Probably the simplest of spatial indices, the grid-based index involves dividing the world into a grid of bins, with each bin containing a list of objects contained inside. If an object happens to cross into multiple bins, each bin includes the object. Because the grid is uniform, simple arithmetic can be used to determine which bins an object potentially occupies, making updates and removals constant speed regardless of the size of the map. For example, in figure 3 below, the red circle could potentially occupy the green and cyan squares. Collision tests place the circle in the three green squares.

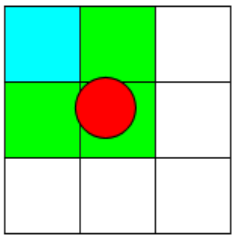

Figure 5: Grid-Based Index

All interactions with the bin index follow the same order. First, to prevent out-of-bounds errors in the array, the shape to be tested is checked for collision with the world box, a bounding box that encompasses the playable world. If the shape is not “colliding” with the world, the function immediately returns false. Otherwise, the array indices of the bins the shape potentially occupies are calculated. For each of these bins, the shape is tested for collision with the bin itself, a box 1 unit wide and deep and extending from the top to the bottom of the map. If the shape collides, it is then tested against the contents of the bin.

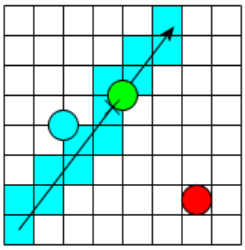

Figure 6: Projectile Collision

While the spatial index and collision detection algorithms are mostly used for testing object collisions, other uses have been found as well. A projectile, a line with the ends designated as previous and current positions, coupled with a target’s ID number can be used to determine line of sight. A spherical cone can be used as awareness volumes, with the related actor or missile being blind to anything not in the cone.

View Frustum Culling

A large fraction of the work the computer does each frame is rendering the scene. Anything that decreases the amount of rendering needed, decreases the amount of work per

frame or, in other words, improves performance. One way to reduce rendering costs is view frustum culling.

The basic concept behind view frustum culling is “don’t render what won’t be visible anyway”. Before rendering each frame, the viewing frustum, the three dimensional region of world space that will show up on screen, is calculated in the form of six planes. Then, before each object is drawn, a rough approximation of its shape is tested against the planes. If the object is determined to be completely outside the frustum, its draw step is skipped. For example, in the following figure, the red circles lie outside the view frustum (cyan) and so aren’t drawn. The green circles are at least partially inside the frustum and would be drawn.

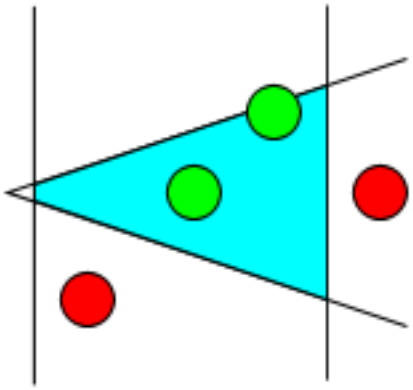

Figure 7: View Frustum Culling

Shadows

Because the monitor can only display a two dimensional image, a problem encountered with all 3D games is the loss of depth perception. There are various solutions used by games; some use 3D anaglyphs to simulate binocular vision. Core13 uses shadows as a depth cue; by

comparing an object's position with its shadow, the player can determine the distance to the object.

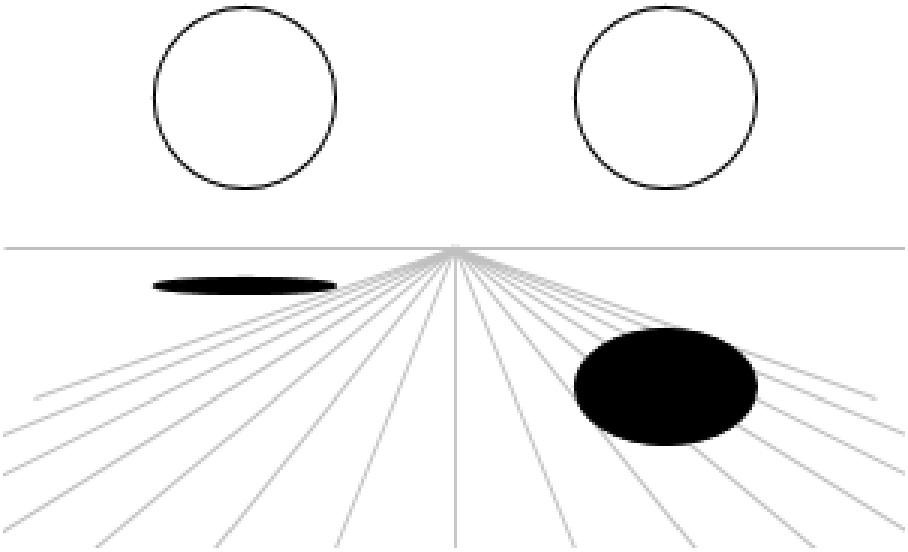

Figure 8: Shadows as Visual Cue

There are a handful of ways to implement shadows, each with their own advantages and disadvantages. The simplest method is projective shadows, wherein the shadow is formed by applying a transformation matrix to the object and drawing it completely black. While this is probably the quickest shadow algorithm, it only works if the receiving surface is flat. Briefly, Core13 used this method; the problems were immediately apparent, as shadows would extend off the edges of buildings and couldn't be cast on moving objects.

A more advanced shadow algorithm used by a number of games is shadow volumes. Each object is given a shadow volume: a shape containing all points that are occluded from light by the object. With each frame, the volumes are rendered into the stencil buffer, and arithmetic is applied to determine which pixels are in shadows and which aren't. While this

algorithm is probably the most accurate, barring the unfeasibly processor intensive ray tracing, the overhead to generate the shadow volumes was a turn off. Either the volumes would be generated with each frame, which would hurt performance, or they would be precomputed, which would require more memory use. Both would require an algorithm to find the silhouette edge of each model.

The algorithm chosen for Core13 is shadow mapping. In shadow mapping, the scene is rendered twice per frame (which makes view frustum culling even more important). The first time, the scene is rendered as depth from the point of view of the light source, and saved to a texture rather than sent to the screen. Then, the scene is rendered in color from the player's point of view. Through shaders, each fragment's light-depth is calculated and compared to the depth value in the texture. If the fragment's depth is greater than the texture value, the fragment is in shadow. The result is proper shading, even on curved surfaces.

Shadow mapping is not without issues. Because the depth values are saved in a texture, the shadows become pixilated. At a distance, the pixilation is barely noticeable, but up close it's obvious. While increasing the size of the texture improves the resolution of the shadows, there's a limit to how big the texture can get. Decreasing the area covered by the shadow map also improves resolution, but then leads to second issue: the texture map can only cover so much space. As a result of that, the dense fog in Core13 is here to stay, to hide the edges of the shadow map. Various solutions do exist to mitigate the resolution and area problems. In Core13, the amount of shading applied to a fragment is determined by checking the fragment's neighbors and interpolating. While this doesn't remove the pixilation, it does soften the edges.

Other solutions range from multiple shadow maps that cover different distances from the player, to additional matrix transformations such that the map has higher resolution close to the player.

Figure 9: Normal View

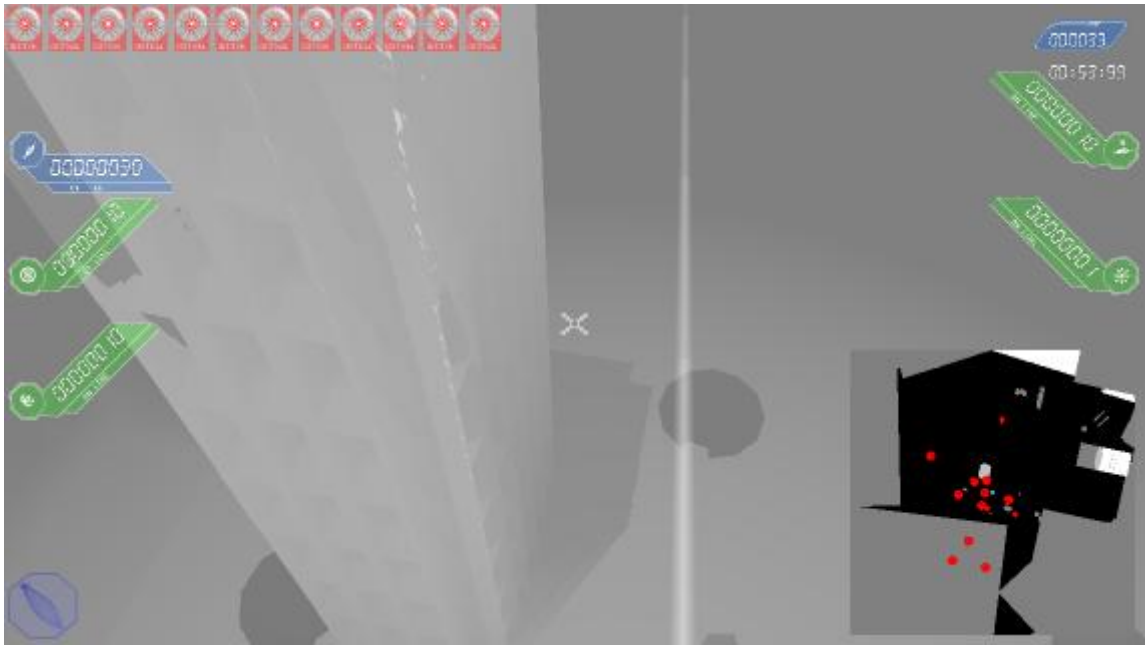

Figure 10: Light Depth read from shadow map. Darker is closer to the light.

Conclusions

Working on Core13 has taught me two things:

- While two or three person teams can work, it's not recommended.
- Be ready to drop features, even if they were part of the original plan. Unless it's required, don't work on a feature unless you know you can make it work perfectly.

Future Work

As mentioned while describing shadow mapping, there are ways of mitigating the resolution problem. Due to time constraints, they couldn't be tried or implemented; however, given more time (476#, anyone?), one of them is bound to work. Aside from that, most of the improvements to be made to Core13 deal with the UI and AI, as well as adding more content.

Resources

www.lighthouse3d.com – provided equations used in view frustum culling

fabiensanglard.net/shadowmapping/index.php – explained how to make shadow mapping work