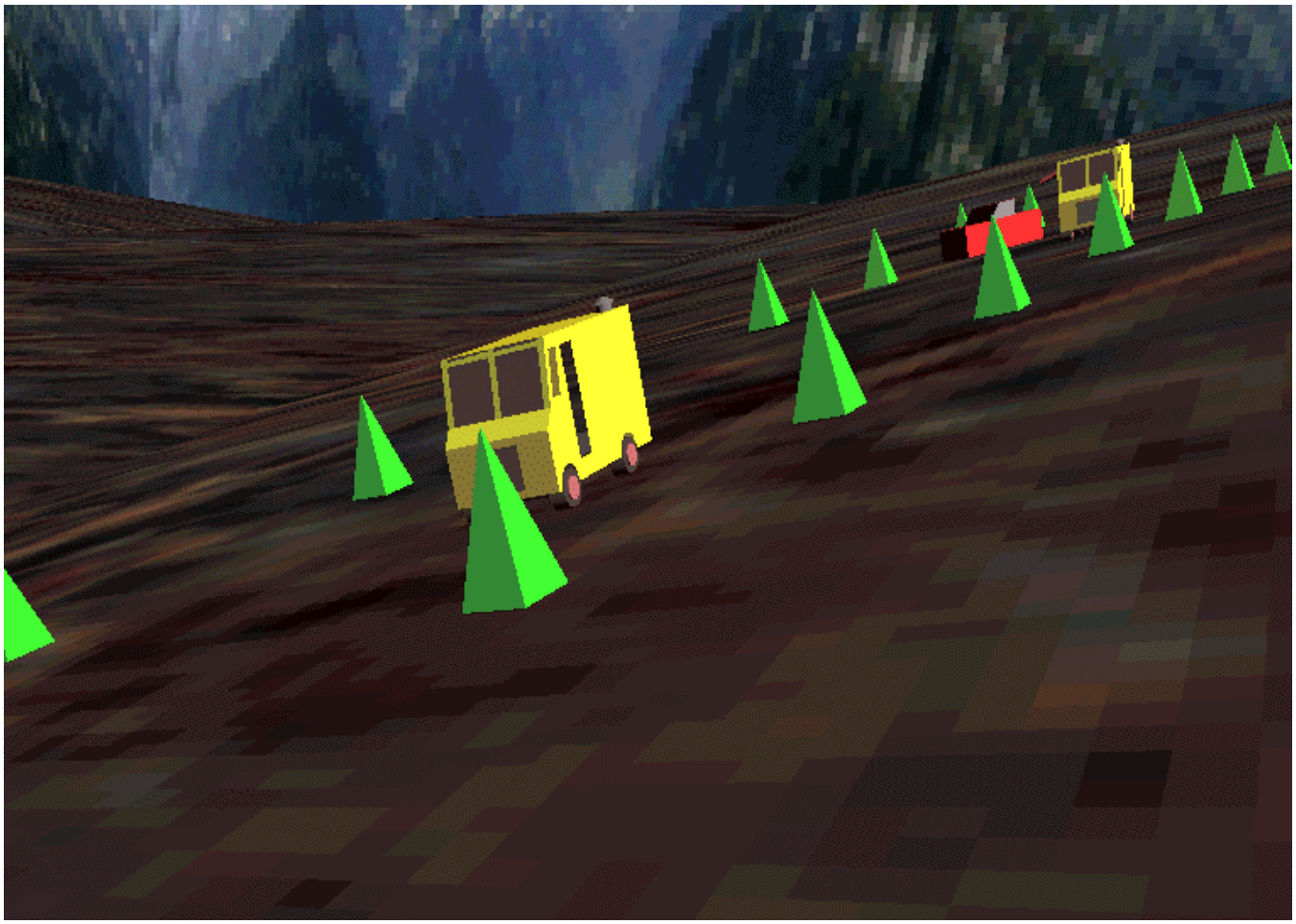


Out of Gas: A Race Until the End

6.837 Computer Graphics Final Report

Team Members:

[Dave Chen] dchen@mit.edu
[Hau Hwang] hhwang@mit.edu
[Jeremy Lilley] jjlilley@mit.edu

Abstract

"Out of Gas" is a racing car game developed in OpenGL and C++. The setting is a hilly, unpredictable terrain with many obstacles, which we render in 3D from a rapidly moving camera that tracks the car. We implemented a detailed physics simulation engine to control the movement of the car and other moving objects across the terrain. This in particular includes height tracking, obstacle collision detection, and accelerations due to gravity. We also included an object-level culling algorithm, which divides the terrain into rectangular partitions and can omit or draw the obstacles in the rectangular patches at different levels of detail depending on distance. Since this is a racing game, we optimized heavily for graphics speed whenever possible, which prompted our development of a proprietary modeling format based on OpenGL, and a suite of related support utilities.

1. ## Introduction

   Over the first part of this semester in 6.837, we explored the graphics pipeline, but for the second part, we wanted to investigate it in a context that is all too familiar to us -- in the context of a game. Specifically, we built a 3D racing car game for our final project, set on a hilly terrain with many obstacles. We wanted to allow a large degree of freedom for the user to drive all over and "see the world" from different peaks and valleys, and we wanted to make sure we could include enough optimizations to allow a decent framerate.

   Throughout our game, there has been a large amount of effort in simulating the movement of the car accurately and being able to track the surface of hills. We deal with variations in height with our height map and physical motion of moving objects with our physics simulation engine. To view this better, we allow the user to dynamically move the camera in relation to the car, so that the player is not constrained to experiencing his collision with a rogue foodtruck from the typical first-person view.

   We implement a number of graphics optimizations to increase the frame rate. This includes rendering obstacles at different levels of detail depending on distance, and culling objects outside the regular field of view. We also use sprites to avoid using many polygons for distant objects.

Our game is written in C++ using the OpenGL libraries, so it can be made to run on many different platforms, just in case not every player has an SGI O2. We use the GLUT toolkit for more flexibility with GL, and by also using the free Mesa3d GL emulation libraries, we were able to target Solaris and Linux workstations as well, and use them in our development.

2. Goals

Over the course of the project, our goals changed as we saw what was more and less feasible to implement in the allotted time. For instance, we expanded the project to better emphasize physics modeling and realistically simulating a car over a rugged terrain with obstacles. Meanwhile, we shifted away from some complex level of detail optimizations that could have easily been projects in and of themselves. With that stated, this is a short list of what we wanted to accomplish in the context of our game.

- **Speed** - Any game needs to be able to render its frames quickly, but in the case of a racing game, this goal is even more critical. We knew this was a necessity from the beginning, and made many decisions based on this, including the use of OpenGL over OpenInventor as our rendering language.
- **Special graphics optimizations**, particularly dealing with level of detail (LOD) - One of the challenges in any outdoor graphics rendering problem is coping with the number of small, distant objects, since there is less natural occlusion than in an indoor situation, which may have many relatively near-by walls that serve as visibility boundaries. There may be many complex distant objects, which while not occluded, may only take up a few pixels on the screen. Our goal was to implement some optimizations that allow us to add more detail and to make the game more realistic. Some of these include:
  - Rendering objects at multiple levels of detail [Shade96]
  - Polygon caching by generating sprites [SS95] - we discovered that this could easily be a final project in and of itself. Nevertheless, we did create sprites for the background and scenery.
  - Using speed in rendering decisions - for lack of a good heuristic of what to drop when driving at high speeds, we chose to concentrate more on object-level culling.
  - Object/higher-level culling - While not explicitly a goal at first, we wanted to avoid unnecessarily sending extra polygons to the graphics libraries. This led to a partitioning algorithm that also became the basis for how LOD is implemented.
- **Physics Modeling** - This was not explicitly a goal at the time of the original project proposal, but as it progressed, the realistic simulation of car movement became one of the more interesting parts of the project.
- **Large variations in terrain height** - We decided that a rugged terrain would not only be more exciting to develop, but it would also allow us to better show off our physics engine. This caused us to place a large emphasis in building a reasonable height map. However, it made other graphics optimizations more complex.

3. Achievements

The main achievement, of course, is producing a game which matches many of our original goals. But that consists of many different parts. Below, we split them into the Game engine, Graphics optimizations, Modeling, and Physics:

1. **Game**

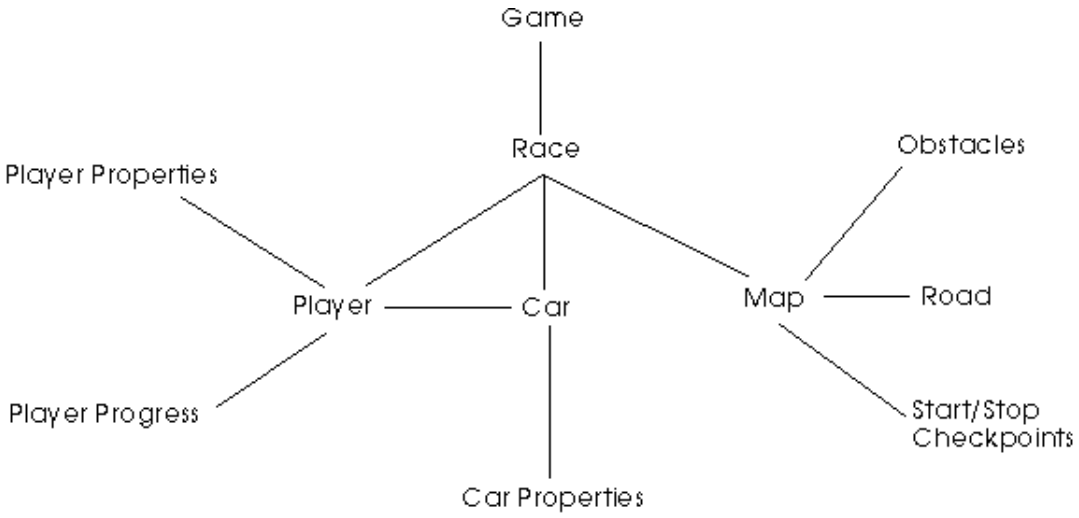
Game engine

Displayed below is a dependency diagram for some of the objects handled by the game. This diagram shows various ‘‘things’’ that are part of a driving game and how they relate to one another. At the top is the Game object, which is the center of the game engine. It sets up the GLUT call-backs, initializes the display, starts making frames per second measurements, and creates a Race object. The Race object creates a Map object, some MovingObjects, and some Players. It then cycles through calling the Physics engine, asking Players to update, and adjusting the camera.

A Player object is conceptually the entity that controls a car object. Associated with a player are Player Properties. This object specifies the name of the player along with the player’s controls (keyboard, AI, network, etc.) A Progress object specifies the player’s state in the race. It keeps track of the laps a player has completed, the current score, and other statistics like the best lap time. Each player is associated with one Car object for a race. The Car object has certain properties as specified in the Car Properties object. These properties include such attributes as acceleration, maximum speed, current speed, location, and other properties cars may have.

The Map object is the other part of a Race. It is the environment within which the Race is to take place. This environment will specify such things as the gravity and weather conditions. The Obstacles object is associated with the Map, and contains information about the various objects in the Map such as cars, buildings, trees, and rocks. The Road object keeps track of all of the surfaces on the Map on which cars can drive. This includes information about how much the surface slows down cars, and how the surface affects a car’s turning radius. The Start/Stop/Checkpoints Object contains information about where the starting line, the finish line, and the checkpoints. Cars must cross checkpoints in the correct order to make progress in the race.

Module Dependency Diagram

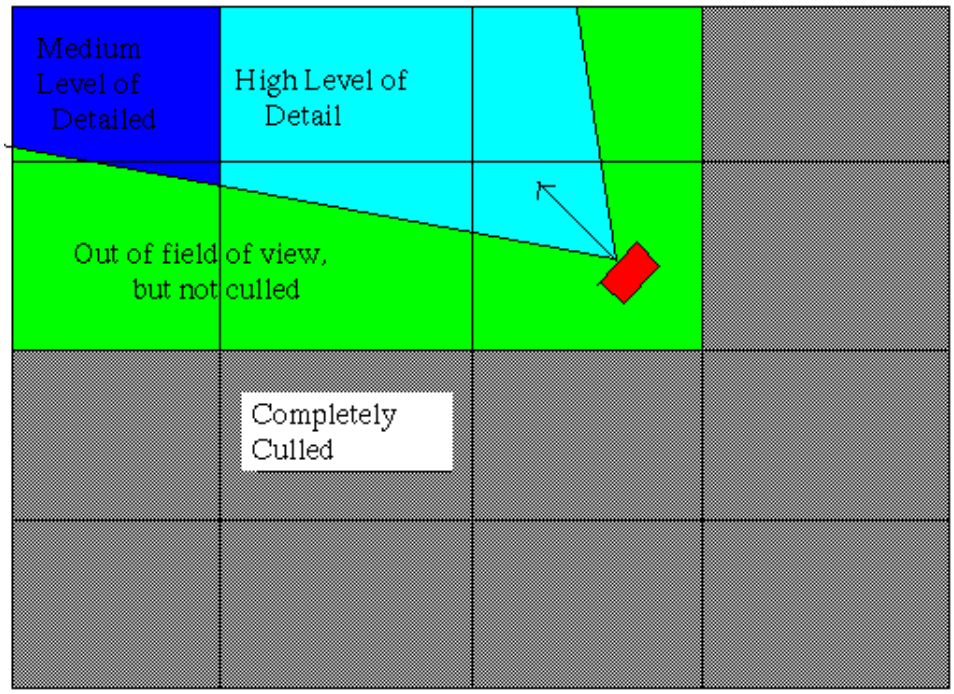


2. **Graphics**

Object-level Culling and Levels Of Detail

Even though OpenGL provides graphics clipping and culling for us, we still wanted to send as few polygons to the GL libraries as possible since speed was a major goal for us. To this end, we implemented an object-level culling and level-of-detail (LOD) engine. We wanted to be able to implement a few high-level visibility tests that would give us reasonable gains and leverage them to increase our frame rate and realism.

The basis of our culling engine is a partitioning scheme. Each map is divided into a number of equal-sized square partitions, usually a perfect square like 16, 49, or 100. Each partition can be displayed in a number of ways: high detail, medium detail, low detail, or completely culled. The level of detail primarily applies to which set of obstacles is displayed in that partition -- we allow multiple models to be used for an obstacle at the same location so that detail of displayed obstacles will vary with distance.



Partition-level culling and level of detail illustrated. The intersection ranges for each partition boundary are calculated, and each partition partially in the field of view is rendered at a different level of detail.

Culling and LOD are based on analyzing which partitions lie inside the field of view, and, of these, how far away they are from the camera. The map drawing routine looks at the vectors for the four sides of the viewing frustum (top left, top right, bottom left, bottom right) and finds the left-most and right-most ones. Originally, we were only analyzing a flat 60 degree (our field of view) plane centered around the direction vector. This would have worked on a flat terrain, but at the strange angles we could encounter on our hilly terrain, there could be a significant deviation from the center 60 degree field of view from the true out-most sides of the viewing frustum.

Once we have the left-most and right-most vector projections, we calculate the intervals along each X and Y partition boundary which the field of view intersects. The intervals parallel to the X axis are calculated by evaluating the equations for the two lines resulting from the field of view vectors going from the camera to the Y intersection points of each

parallel. The three resulting regions are tested with plane inequalities derived from the vectors, which will show which interval corresponds to the true field of view interval. Points behind the camera will have no valid interval. The same is done for each division parallel to the Y axis.

After the intersection intervals are calculated, the partitions can be displayed. If one of a partition's four sides intersects the field of view, as determined from the pre-calculated intervals, it may be displayed. But beyond that, the culling engine also calculates the minimum perpendicular distance to the nearest edge of the partition. Distance thresholds are established and used to determine whether the entire partition is displayed at a high, low, or medium level of detail.

The reason for the intersection intervals is that we found this to be the easiest way to do correct field of view culling. At first, we were only examining the four edge points of each partition, but this was problematic close to boundaries when the camera's field of view did not include any of the near partition's vertices.

To mention scalability briefly, one of the main motivating factors of the partitioning scheme was to allow the terrain system to handle larger files than it normally could have. Each partition represents an independent fraction of the entire world and helps us to scale the system. The divisions of the terrain and roads are pre-computed, using a set of automated scripts which we developed in PERL. When the pre-divided terrain is loaded, each terrain portion gets put in a separate partition list. Obstacles are similarly placed into different partitions when they are read from the map file. Each partition holds a linked list for each of the three levels of detail. If an obstacle straddles a partition boundary, it is placed in both partitions' linked lists, and a sequence number will be used to ensure that each object is only drawn once per map refresh.

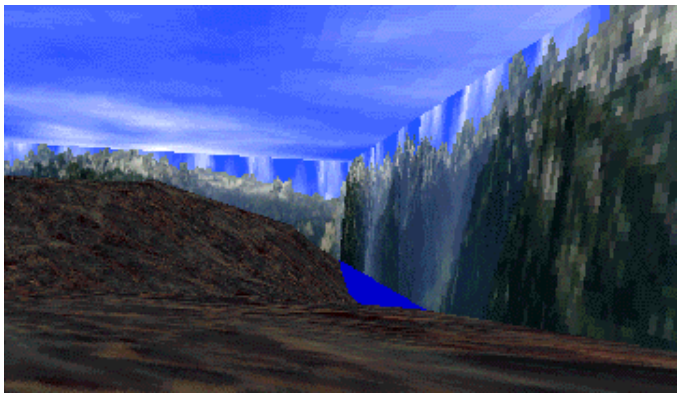
Inside our Map file, as described below, we specify for each obstacle on the screen at which levels of detail it is visible. Thus, more detailed images can be displayed close-up, and less detailed images can be displayed far-away. In addition, fewer bulk objects, such as trees, can be displayed at further distances, when they are less noticeable. During the process of our design, we went through different numbers of levels of detail. At the beginning, we chose five, but found that this was way too many. In reaction, we oversimplified to two levels: "low" and "high," but feared "popping" would be too much of a problem. We finally settled on three levels of detail, and it seemed to be sufficient for the purposes of our game.

Before arriving at a partitioning scheme, we were contemplating using a BSP tree. But, we decided that using partitions would give us the ability to do object culling without requiring the same implementation complexity. Unfortunately, we still ended up having to implement the polygon splitting that would be required for a BSP tree anyway, but we were able to do that inside our splitobj.pl PERL script, which was probably less complex than implementing it in C.

And, in any such partitioning scheme, there is a fundamental trade-off between finer detail control and the amount of overhead from these partition checks. The main reason for this trade-off is that border partitions need to displayed even even if the intersection area is small. Making the partitions smaller will reduce the wasted space, but cause more related

processing overhead. We have not precisely quantified this trade-off, since it is highly dependent on the models on the screen. But, we feel, given some of our profiling numbers, which show the graphics drawing calls taking roughly 100 times more processing time than all of our physics update calculations, that we can afford to invest a fair amount of CPU time for finer grained culling to reduce the number of OpenGL calls. Most of our development has been with a relatively low number of partitions (sixteen), but we can easily scale this number to do finer level checking.

Sprites



Illustrating the use of sprites for the "sides of the world" and sky.

We used sprites in various places to avoid having to render an equivalent scene with massive numbers of polygons. This was done with texture-mapped polygons, disguised in the map as Obstacles. At first, having read a conference paper [Shade96], we hoped to be able to generate these sprites on the fly from polygons, and use sprites more as a cache. However, we found that this could itself be a final project. In addition, our rugged terrain would have made sprite-based polygon caching more difficult to implement, since the car can be at very strange z angles from the objects. We found these edge sprites to be sufficient to reduce unnecessary polygons.

File Formats, Conversion

We used OpenGL as our graphics API, but wanted to be able to use higher-level graphics tools, such as OpenInventor, Alias Wavefront, and TrimNURBs, as modeling tools. To facilitate this, we created our own ASCII graphics file format, which we termed the *Object* format, that is tightly coupled with various OpenGL commands. There is an appendix outlining this format.

There are a number of steps for importing models for use in our game.

1. Each incoming OpenInventor file must be processed by *ivfix* , which converts all polygons, spheres, etc. to triangle strip sets. These pre-rendered triangles are more efficient for subsequent layers to process.
2. Our *conv.pl* PERL script converts the subset of OpenInventor created by *ivfix* tool into our Object file format. Most of the changes involve taking out braces and using regular expressions to change the object-oriented Inventor format into our easy-to-parse OBJ file format.
3. Terrains require an additional step. Since a terrain is created as one large piece, it must be broken along the boundaries of the various partitions. At first, we thought of not splitting triangles, but including every triangle with a footprint in the partition, but this seemed inefficient. Overlap also defeats many of the benefits obtained in true partitioning. We created another PERL script, *splitobj.pl*, to do this clipping, using a window clipping algorithm obtained from our 6.837 book. This proved to be more difficult than we anticipated at first, but using it allows us to design our terrain in a

manner very independent of the partitions.

Fonts

Since OpenGL does not have a way of displaying text on the screen, we wrote our own routines to do so. OpenGL has a mechanism for displaying bitmaps, but creating a bitmap by hand for each text character that we want to display would have been very tedious. To generate the bitmaps automatically, we found and used a tool called "fontconvert", which is part of the "fontutils" package. This program reads in PostScript font files and outputs the font data in several formats, one of which is a plain ASCII text format. Out of Gas parses this output for bitmap data, as well as the height, width, and placement information which is associated with each character. With this mechanism in place, we are able to display text from any PostScript font files.

3. Modelling

The car's driving environment is composed of several parts. Obstacles are static items, such as trees and parked food trucks, which are placed on the terrain. The terrain consists of the models for the ground and track. The MAP file is used to "glue" the Obstacles to the Terrain.

Obstacles

The static objects on our maps, not including the ground, are known as Obstacles. Trees and foodtrucks are good examples of these. For the most part, they are modeled in OpenInventor and converted to our proprietary Object file format using PERL scripts as mentioned before. An OpenGL display list is compiled as each individual Object file is load. All the Obstacles that use that same modeled Object (i.e. all trees on the screen) can share the same single common Object. State is kept for each Obstacle so that it knows its own orientation and location on the map. All obstacles are modeled in the positive octant, as mentioned in the Modeling Conventions appendix.

Terrain

Generating a terrain without spending a lot of effort modeling it is difficult. We initially spent some time building our own terrain modeler, where the user interacts with the terrain, specifying where to add or remove polygons. However, we soon hit upon the idea of using NURB surfaces to generate the terrain. With NURB surfaces, we can create a varied and interesting terrain with very little effort. We then use "ivfix" to turn this NURB surface into polygons. This tool, which is part of the Inventor toolkit, will render an Inventor file to its component triangles. By using a "complexity" node, we can adjust the number of polygons that we spend on the terrain.

We decided to do this conversion to store and render the terrain as a collection of polygons instead of directly using a NURB surface representation for two reasons. First, we assumed that rendering the terrain as a NURB surface would be too computationally intensive for our purposes. Second, Mesa's NURB support is sketchy.

Using a NURB surface to generate the terrain presented several problems. Texturing the surface with more than one kind of texture is difficult. One possibility for texturing the

surface was to use one very large texture which spanned the entire surface so that each polygon in the surface uses only a small portion of the large texture. This has the advantage of our being able to use a paint program to generate what the terrain should look like, and then plastering this over the terrain. "ivfix" generates the corresponding texture coordinates which are used by Out of Gas, and so the surface is rendered correctly. However, to avoid having a blocky looking terrain, the texture has to be unmanageably large. The other option for texturing, which is to texture each polygon with an entire texture, works well, except specifying which polygon gets which texture is difficult. To resolve this problem, we use trimmed NURB surfaces, using the NurbsProfile nodes in Inventor. With trimmed NURB surfaces, we generate multiple surfaces, each of which can be textured with a different texture map. Since we use trimmed NURB surfaces, we do not have to worry about trying to create multiple NURB surfaces which fit together. We use Professor Teller's "trimnurbs" program to generate our terrain surfaces, and translate the output from this program into Inventor.

One problem with this approach is that adding small details to our terrain is still difficult. For example, there is no good way for us to draw the lane markings on roads.

MAP file

We created a Map file format to specify the placement of different obstacles on the terrain. This essentially serves as the "glue" for the obstacles that compose the world. A specific description of it is available in the appendices. It is designed as a text file, and has a number of entries, including the following:

- The Dimensions of the scene and number of partitions
- Which Terrain file and Height Map to use
- Each Obstacle, including
  - The Object to use (car, tree, truck)
  - X, Y, Z coordinates
  - Whether the Z coordinate is absolute or relative height from the ground
  - The levels of detail that it should be rendered at, which are "high," "medium," and "low"
  - Whether or not collision checking should be used on the object
  - Rotation from the original position

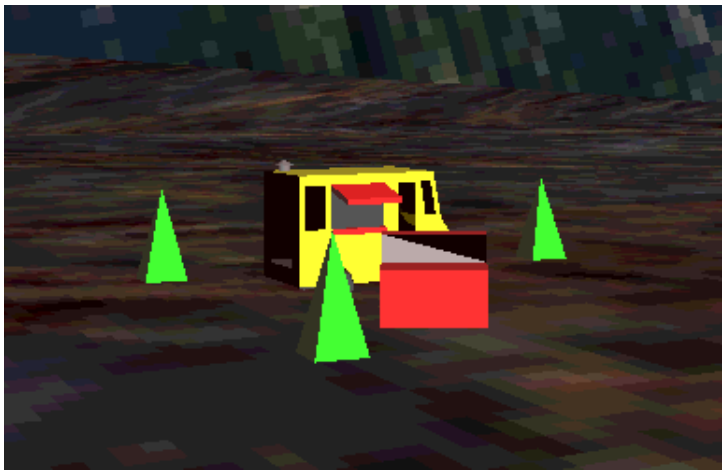
4. **Physics**

Collision Checking

Collision checking between objects in the game needs to be fast and efficient for the physics engine to run smoothly. Collision checking occurs in two stages. The first stage performs rough bounds checking on the objects to see if there may be potential collisions. Objects which pass this test undergo a second stage of collision checking which determines whether a collision actually occurred.

Coarse collision detection involves performing spherical bounds checking between objects. Given the location of two objects to check, our algorithm determines the straight line distance between the two and checks whether this distance exceeds the sum of the radii of

the spherical bounding shells for the two objects. If the distance is greater, then we know the two objects are nowhere near one another and cannot possibly collide. If the distance is less, we run the more fine grain collision test to check for collisions.



Illustrating a car being stopped by collision checking when hitting a food truck.

Fine grain collision checking involves determining whether there are intersections between the bounding boxes for two objects. Since the bounding boxes for our objects are very tight, the intersection of an object's bounding box with another will indicate the presence of a collision. A quick way to detect an intersection is to see whether or not each of the vertices of one object's bounding box is contained within the second object's bounding box and vice versa. It turns out that this simple check is sufficient for detecting all collisions in our game.

Performing this fine grain collision detection naively could be very inefficient as well as challenging since the various objects on the map have probably been rotated, translated, tilted, etc. thus causing their bounding boxes to be in all sorts of different orientations. We solve this problem by performing a change of basis on the world space coordinate of the point we would like to do bounds checking on. This point is transformed into the space which has as a basis the length, width, and height vectors which determine the bounding box of the object which we want to check against. Let the matrix M have the length, width, and height vectors of the bounding box in its columns. Let c be the coordinate of the point we wish to transform into the space with basis equal to the columns of M. Then all we have to do to find d, the coordinate of the point c in the new space, is to take d=(M^-1)*c. Note that because M has orthogonal columns, we almost have M^-1 if we just take the transpose of M. In fact, if we just calculate d as shown above using M transpose in place of (M^-1), we can just compare each dimension of d against the squares of each dimension of the bounding box (the square of the length, width, and height) to determine whether or not the point is contained within the bounding box. This only involves six comparisons since we compare the corresponding dimensions of d against the square of the length, width, height, and also against length=0, width=0, and height=0 (recall that the bounding boxes of all our objects always have one vertex at (0,0,0) in object space and the rest of the bounding box lives in the octant with all positive coordinates - see Modelling Convention Appendix). This procedure allows us to quickly perform collision detection.

Physics updates

The physics engine cycles through each MovingObject on the map, and updates the environment according to their state. For each MovingObject, the physics engine calculates new velocity vectors, new locations, and new direction vectors. The velocity is calculated

based on the MovingObject's old velocity vector, gas and steering settings, gravity, and collision detection. The new location is dependent on the velocity, height map, and collision detection. The new direction vector for each MovingObject is calculated based on steering settings and the velocity vector.

To determine the new location the physics engine attempts to move the object according to the velocity vector. After doing this, the car may be beneath the terrain, and so wheels that are beneath the terrain are moved to be on the terrain, according to the height map. After moving these wheels up, the MovingObject's location and orientation are set by intelligently picking three wheels, forming a plane out of them, and using this plane as the bottom plane of the MovingObject. After adjusting for the terrain, the MovingObject may still have collided with some other object. Collision detection is then performed, and the MovingObject is moved to be flush with any objects with which it has collided.

Height Map

In any 3-d game, the problem of specifying location along the z or up direction must be addressed. To help solve this problem, our game made use of a height map. Specifically, a height map allows us to quickly obtain the z coordinate, or height, of any point on our terrain quickly. This is very useful because it allows our car to drive on bumpy terrain and facilitates the placement of models on the map.

We dynamically generate a height map from the terrain file each time we load a new map. To create a height map, we sample the heights of the terrain along the x and y directions at even intervals starting at x = y = 0. For example, if the terrain had x and y dimensions of 400 by 500 meters and our sampling interval was 5 meters, then our height map would contain (400/5 + 1)*(500/5 + 1) points. To determine the heights at a particular x y location, we first determine which particular triangle from the terrain contains the point. This is done by plugging the x y point into the plane equations that form the edges of a candidate triangle. If the x y point is contained by the triangle, we just plug the x y point into the plane equation of the triangle and solve for the height z. We iterate this procedure until we obtain heights at all the points we wish to sample at and store these heights in an array.

Once this height map array is generated, we can quickly determine the approximate height of any x y map position quickly. The first step is to determine the closest three points on the height map to the specified x and y position. This is pretty much a constant time array lookup into our height map. From these three points, we perform fast linear interpolation to find the desired height at position x and y.

Our height map can be tuned to suit the needs of particular maps. Specifically, we can choose to sample at smaller intervals so that we get greater fidelity to the actual heights of the terrain. This may be useful if the terrain is very hilly or rocky. The tradeoff here is that by decreasing the sampling interval, we increase the height map's memory footprint. On the other hand, if the map is very flat, we can have the height map take up virtually no memory at all by specifying a very large sampling interval. So our height map is flexible enough to suit the needs of particular maps.

Overall, we felt the use of a height map was a big win for our game. By performing a bit of

precomputation to generate the height map as we load the game map, we are able to obtain the terrain height at any x and y location quickly.

4. Individual Contributions

We worked on the project in a team capacity, but, like any team, we individually focused on different aspects of the project:

- Dave concentrated on the game engine, and worked on the physics engine.
- Hau spent much of his time on the physics engine and height map, as well as finding reasonable ways to model a rugged terrain.
- Jeremy concentrated on many of the graphics optimizations, and the graphics format converter scripts.

We all spent some time modeling, testing, and debugging.

5. Lessons Learned

Building a racing game was quite an learning experience. We learned about GL coding, culling, game design, and how to build physics and game engines.

One lesson we learned was that we probably should have used the Inventor api so that we could have directly used inventor models in our game. This would have saved us a lot of time and effort since we ended up writing converters from Inventor to our own modeling format to be displayed under GL.

We also found out that building a realistic physics engine takes a considerable amount of time. It takes a lot of careful thought and effort to make objects in the game behave like their real world counterparts.

There were other parts that seemed to be simple at first, such as clipping a set of polygons along rectangular boundaries for partitioning, which turned out to be more work that we thought.

As with any large project, we quickly learned that good modular design is essential. We worked on several versions of the physics engine and if we had not abstracted the engine from the rest of the game, it would have been very difficult to revamp the engine each time.

6. Acknowledgments

We are grateful for the helpful comments from Professor Teller, particularly related to our initial project proposal, and we appreciate the hints and advice from our TA, Kari Anne Kjolass.

Tools we used:

- Mesa3D 3d library
- TrimNurbs
- Open Inventor
  - ivview
  - ivfix
- The GIMP
- XV
- Alias Wavefront
- GNU Emacs
- Netscape
- readtex.C (RGB code) from the Mesa code
- Linux, Solaris, Irix

Jason Yang from the graphics group gave us hints and lent us a number of graphics books.

The people at LaVerde's and McDonald's who sold us food when we took breaks from 4-035.

7. Bibliography

1. **[directx]** http://www.microsoft.com/directx
2. **[www.mesa3d.org]** *The Mesa 3D Graphics Library* http://www.mesa3d.org .
3. **[www.opengl.org]** http://www.opengl.org
4. **[HB96]** Hearn, Donald, Baker, M. Pauline, *Computer Graphics C Version* , Prentice Hall, 1996.
5. **[Lam95]** LaMothe, Andre, *Black Art of 3D Game Programming* Waite Group Press, CA, 1995.
6. **[SS95]** Schaufler, Sturzlinger, *Generating Multiple Levels of Detail for Polygonal Geometry Models*, EGWVE 95.
7. **[sgibsp]** *BSPTree Frequently Asked Questions*, http://reality.sgi.com/bspfaq .
8. **[Shade96]** Shade, Lischinski, Salesin, DeRose, and Snyder, Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments, SIGGRAPH 96.
9. **[Wern95]** Wernecke, Josie, *The Inventor Mentor*, Addison-Wesley, Reading, MA, November 1995.
10. **[WCS96]** Wall, Christiansen, and Schwartz, *Programming Perl*, O'Reilley, CA, 1996.
11. **[WND97]** Woo, Mason and Neider, Jackie and Davis, Tom, *OpenGL Programming Guide: the official guide to learning OpenGL, version 1.1*, Addison-Wesley, Reading, MA, January 1997.

8. Appendices

1. **Basic modeling conventions**

We decided we needed a set of modeling conventions from the start to ensure interoperatability of our models and prevent easy mistakes. Here is a brief summary:

- The Z vector represents "up," while the X and Y vectors form a corresponding right-handed coordinate system.
- One unit in our modeling conventions represents one meter. Each object is to be modeled with that in mind (so a car might be modeled within a 3x5 area while a tree might be 1x1).
- The X vector represents "forward" in our untransformed object models, such as cars and foodtrucks.
- All objects are modeled with positive coordinates only, and start with their lower-right point at (0,0). They extend as far as (xSize, ySize).

2. **Map file format**

The Map file format specifies characteristics of the map, including where obstacles are initially placed. The format is free-form ASCII, with C-style `/* */` comments allowed.

Commands:

Partitions [x_par] [y_par]
: Partition the map into `x_par` partitions in the X direction and `y_par` partitions in the y direction. This specifies the granularity of the cells used in culling.

heightMap [filename]
: Use the specified filename as the heightMap.

terrainMap [filename]
: Use the specified filename as the terrainMap.

obstacle [obs_type] [x_pos] [y_pos] [z_pos] [ABS/REL] [theta] {L}{M}{H}{C}
: Place an obstacle on the map. The `obs_type` specifies which kind of obstacle to be added. These obstacles can be trees, cones, cars, etc. The string used for `obs_type` is specified in `StdDefs.h`, where this a mapping from these strings to the Object file format files which contain information about the object. `x_pos`, `y_pos`, and `z_pos` specify the position of the obstacle. `ABS` indicates that `z_pos` is an absolute distance, measured in world space. `REL` specifies that the `z_pos` is an offset from the heightMap specified height at the given `x_pos` and `y_pos`. `theta` specifies how far the object should be rotated around the `z` axis for its orientation. The `L`, `M`, and `H` flags specify which levels of detail the object should be displayed at, corresponding to low, medium, and high, respectively. For example, if only `M` and `H` are specified, the object will only be rendered when the graphics engine determines that the object is at such a distance that it should be drawn in medium or high detail. The `C` flag specifies whether or not the physics engine should perform collision checking between the object and moving objects.

3. **Object file format**

We decided not to use OpenInventor due to speed considerations, yet we still wanted a method using files for reading modeled objects into the game. One approach would have been to generate C++ code to render an object in OpenGL, but instead we chose a more

compact representation that has a relatively simple mapping to various Open GL calls.

The format is free-form ASCII, with C-style `/* */` comments allowed. The first line must be "a", to stand for ascii.

Commands:

glNewList [objectName]
: Create a display list for the objectName. (All the GL commands must be run between a glNewList and glEndList to be stored anywhere.)

glEndList [objectName]
: Ends the recording for the display list. The objectName must start the objectName for the start of the display list.

glPushMatrix
: Saves the state of the transformation matrix. Similar, but not completely analogous to the Separator in Inventor (since glPushMatrix only deals with the transformation matrix).

glPopMatrix
: Pops a saved matrix transformation.

Dimensions [x y z]/OverallDimensions [x y]
: Specifies the dimensions of the object. In partitioned files, `OverallDimensions` is the total size, while `Dimensions` is each partition's size. Unpartitioned files have the same `Dimensions` and `OverallDimensions`.

glMaterialfv [side] [type] [r] [g] [b] [a]
: Allows the material to be specified. [side] may be one of: { `GL_FRONT, GL_FRONT_AND_BACK, GL_BACK`} Type may be { `GL_DIFFUSE, GL_AMBIENT, GL_AMBIENT_AND_DIFFUSE, GL_SPECULAR, GL_SHININESS, GL_EMISSION` }

glVertexPointer [number] [...]
: Adds a number of vertex points into the resident vertex point state. [Number] species the number of [x y z] points are ahead to be read.

invIndices [vertices -1]* -1
: Specifies how the vertices get connected to form polygons. A -1 ends each polygon, and a final -1 ends the sequence. For example, to make a polygon with vertices 0 through 3 and another polygon with vertices 4 through 7, the sequence would be `0 1 2 3 -1 4 5 6 7 -1 -1`

invTriIndices [vertices -1]* -1
: This is the same as invIndices, except that the points are interpreted as GL triangle strips.

flushIv
: Draws the set of indices specified in invIndices/invTriIndices, along with the appropriate store texture and normal values, if they are specified as mentioned below.

glEnableClientState/glDisableClientState [type]
: These remain part of the specification language to mention when whether the vertices, normals, or texture coordinates are to be specified. But, when used on the output of ivfixed files, these directives are not strictly necessary.

glNormalPointer [number] [...]
: Specifies a list of normals to be used with the vertices. Note that unlike in Inventor, there is no mapping between the normal at place X in the list and the vertex at place X

in the list. If not used, Normals will be manually calculated by assuming vertex points are specified in counter-clockwise order.

invNormals [vertices -1]* -1

Specifies a list of normals to be used with the vertices, in the same manner of invIndices. These two must have the same length, but are independent so as not to require the list of vertices and list of normals to be coupled.

glBindTexture [textureName]

Specifies the texture to be used in a subsequent texture-mapping operation.

glTexCoordPointer [number] [...]

If this is used, it specifies the texture coordinates (x y) that can be used with vertices.

invTexCoords [vertices -1]* -1

Like invNormals, this will list out the texture coordinates from the glTexCoordPointer list that correspond to the points in invIndices/invTriIndices.

The glVertexPointer / invTriIndices / flushIv method is the preferred way to generate objects. However, we also support the following calls, which are comparable to the analogous OpenGL calls:

glBegin [type]

Specifies the beginning of a object set, for which the glVertex3f calls can be issued. [type] can be one of { `GL_POLYGON, GL_QUADS, GL_QUAD_STRIP, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP` }

glEnd

Specifies the end of an object set.

glVertex3f [x y z]

Specifies a vertex to be drawn to screen.

glNormal3f [xhat yhat zhat]

Specifies the normal to be used on subsequent operations.

glColor3f [r g b]

Specifies the color to be used on subsequent operations.

glTranslatef [xDiff yDiff zDiff

Translates the transformation matrix.

glScalef [xScale yScale zScale]

Scales the transformation matrix.

glRotate3f [amount x y z]

Rotates the transformation matrix.