Going Nowhere Fast: A 3-D Driving Game

6.837 Final Project

Stanley Hu, Matt Lee, Chad Talbott, Jordan Weitz

Abstract

Going Nowhere Fast is a basic driving game set in a dynamically generated 3-D city, complete

with texture mapped structures to provide an attractive look. The user can choose from several types of car models, drive them around the city, and see the car respond to physical forces. Various different cameras are provided to view the world, while a simple computer-controlled car can be found driving around the city.

Introduction

Going Nowhere Fast draws its inspiration from popular video games where a user can indulge his worst instincts by ramming into other cars while driving around the city. It is the product of four weeks of seemingly endless discussion, coding, and hard work. It was also the source of many bad jokes.

Given the allotted time, we think we have created a visually stunning game that provides many interesting features for both the novice and graphics-minded audience. The dynamically generated city makes "no game the same", while the physics engine gives the car the ability to react to external forces. We have implemented a visibility detection algorithm to optimize the speed of the game and provided many different cameras to allow viewing from a variety of angles.

In the end, we hope someone will be entertained by *Going Nowhere Fast*-so much so that we can sell it and retire after this term.

We hope you enjoy it.

Goals

When we set out to design *Going Nowhere Fast*, we wanted to create a fun 3-D game where a user could drive around and ram into things for no apparent purpose. To provide an interesting backdrop for the game, we chose a city as the scene (as opposed to a race track). Our hope was to have the user awed by the beauty of the surrounding structures while driving around. Cars would respond somewhat realistically to physical forces, such as gravity and acceleration, when it hit a curb or another object. In addition, we wanted to give the user the ability to change viewing angles as he played the

game. Lastly, we wanted a number of computer-controlled cars that would be out to "attack" the user as well.

In summary, our goals included:

- Creating a visually interesting 3D world
- Modeling physical effects including gravity, acceleration, and other forces
- Providing a wide selection of camera angles to illustrate various views and perspective transformations
- Creating an entertaining, interactive game for generations to enjoy

Design

In our first week, we created an object dependency diagram for our code that would serve as our initial design. In this diagram, we showed how each module would interact with another through function calls. The following diagram shows the results of our initial design:

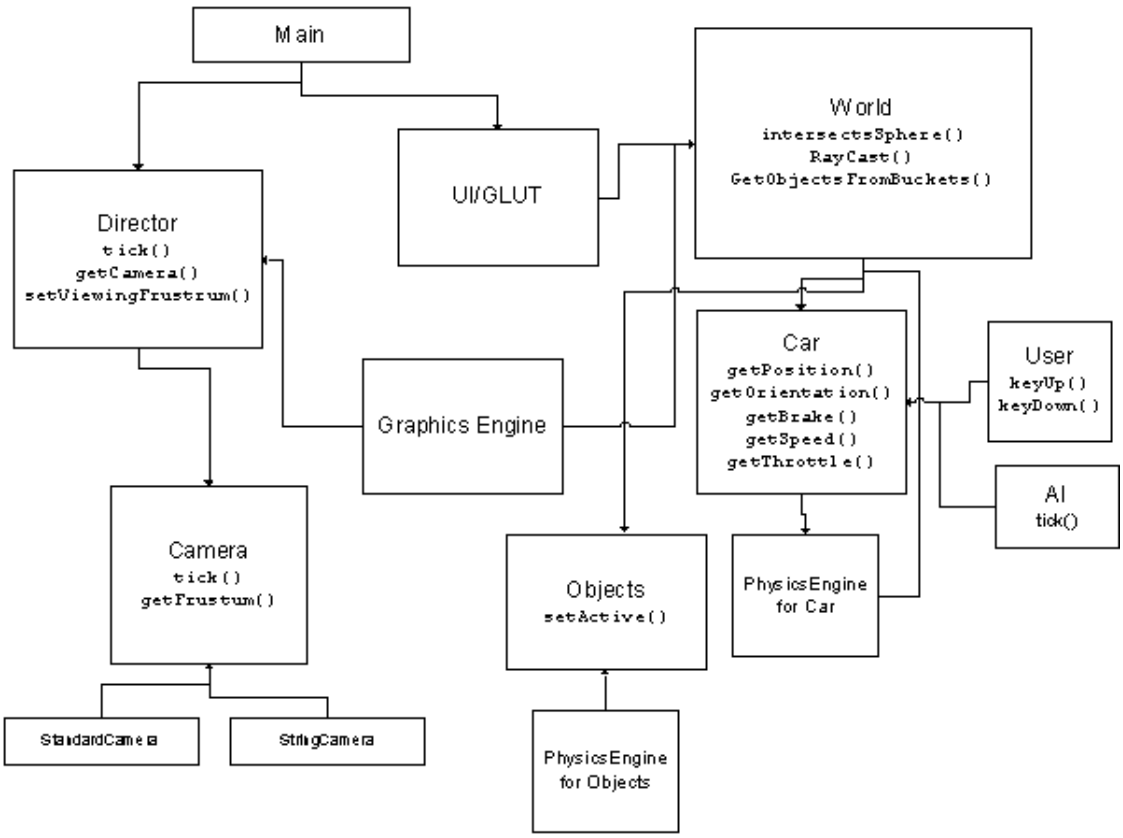


Figure 1. Initial design diagram for *Going Nowhere Fast*.

With this diagram, we were able to describe in detail how the program would work. First, the *Main* class would be responsible for loading the program. It would take

user arguments from the command-line, parse them, and initialize the *GraphicsEngine*.

Next, the *GraphicsEngine* would load and create the world geometry, which would consist of a series of tiles with buckets of polygons. It would then create a *Director* object, which would be responsible for choosing which camera view to use. The *Director* object would initialize the cameras it needed and provide methods for updating state information. Ideally, we wanted the *Director* to compute the "best" camera angle given the car's location. However, to simplify things, the *Director* simply chooses the camera based on user input-camera angles can be changed with the use of the function (F1-Fx) keys.

The *GraphicsEngine* is also responsible for updating the state of the world as time elapsed. It has an idle loop that calculates the amount of time that has elapsed and updates the objects in the world accordingly.

In our design, we rested on the assumption that objects that needed to be drawn would contain a *draw()* function, and the GraphicsEngine would simply call that method. Since state information needed to be updated, they would also need a *tick()* function that would be called as time elapsed. Soon, we found that we needed to provide an interface to make it easy for the *GraphicsEngine* to iterate through the objects that needed to be drawn. A *DrawObject* interface, consisting of virtual methods, was created to support this.

The *GraphicsEngine* also initializes the cars and places them in the world. Since cars implemented the *DrawObject* object interface, they would be responsible for drawing themselves. This allowed us to easily add or change car models when we desired.

Each car also has physics engine that calculates performs operations such as updating the car based on the effects of acceleration, braking, and turning. In our design, we wanted to have other inanimate objects, such as garbage cans, that would also be affected by physics. However, many of the physics calculations were really specific to cars. This is why the diagram shows separate *PhysicsEngine* objects for cars and objects.

Achievements

We were able to create a simple game where a user can drive through an esthetically attractive 3-D city with roads, complete with a variety of structures, such as skyscrapers, churches, and parking garages. We modeled braking, steering, and acceleration forces with a simple physics engine, and we provided a series of cameras with which to view the game. In addition, we implemented a simple visibility detection algorithm and artificial intelligence (AI) for a driver.

Our game runs on a variety of platforms: Solaris, IRIX, and Windows. We had to tweak code numerous times to make this possible.

Artwork generation. Among the first challenges we encountered were how to create the models for our city, loaded them into our program, and make them look attractive. We first created the city tiles with 3D Studio MAX. Special names were assigned to each object based on the desired texture for each surface as well as the type of object that was being named; for example, buildings and curbs

were assigned different names. The geometry was then exported to 3DS format and then converted to RAW format using a utility called 3Dto3D.

We chose to use the RAW because it was an easy-to-parse ASCII format. However, after we noticed that the RAW format failed to preserve texture coordinates, Matt wrote a program called TexBuilder to regenerate proper texture coordinates for the geometry.

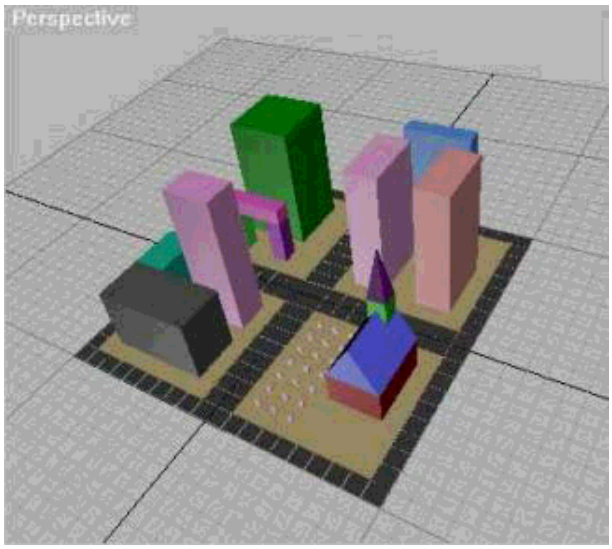
TexBuilder uses the descriptive names assigned in 3D Studio MAX along with the geometry data itself to infer proper texture coordinates for all of the tile geometry. This data is placed in a RTX file, which corresponds to the RAW file line by line. A RTX file specifies texture coordinate and color information for one polygon per line, similar to the RAW format. The world loader reads the RAW and RTX files for a tile in parallel when reading in a tile.

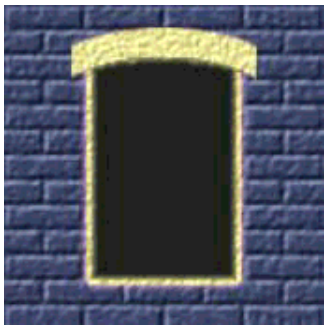
Textures were generated from scratch in Photoshop 5.0. They were saved in Targa (TGA) format, which is a raw image format that is easily loaded into the application. Based on command-line options, GL textures of various types (nearest, bilinear, trilinear, mipmapped, etc) are generated from the TGA files.

Figure 2 shows a sample model of a city block. Figure 3 shows a texture that will be applied to the building.

Figure 1 (left). A city tile modeled in 3D Studio MAX.

Figure 2 (below). A texture used for a building.





We also created models for cars. We first modeled them in AC3D and used a program to convert the models into OpenGL code. Ideally, we would want a way to dynamically load the car models, but we chose this approach to save time.

The user can select from three different car models: a simple, yellow car; a red, more detailed sports car; and a blue limousine. Figure 3 shows the car models.

Figure 3. A picture of the user-selectable car models.

Graphics Engine Our graphics engine creates the city at runtime, so that the city will usually be different from the last time the program ran. Each of the tiles contains textured-mapped structures with our generated artwork. Careful examination of certain textures reveals some interesting details.[1]

Each tile, or city block, is dynamically loaded at runtime, making it very easy to add or remove new kinds of structures. Adding a new tile is a simple matter of taking its model, converting it to our format, and then letting the graphics engine load it.

A tile is a list of texture-mapped polygons that make up a section of the world. Each tile is the same size in the z = 0 plane; a square from (-100, -100, 0) to (100, 100, 0). There is no upper bound for the height of the tile, but most tiles are no taller than 200 units.

The world loader parses a city definition file (city.gnf) that specifies which tiles to load. Each tile is loaded into a linked list; the collection of tiles is stored in an array of pointers (master tile list). The city itself is stored in another array of pointers; each item in the array is one of the tiles in the city, and it points to one of the tile lists. Therefore, the master tile list maintains a 1:1 relation with the tiles, while the city array may have multiple items pointing to the same list. There is also a parallel array of integers for the city array; it maintains the tile rotation, which is a multiple of 90 degrees.

To render the city, each tile in the city tile array is transformed to the correct location and drawn. To ensure that the frame rate stays relatively constant across various city dimensions, each tile is tested against the view frustum before it is drawn.

Visibility Detection We implemented a simple algorithm for visibility detection to optimize the speed of the program. This algorithm decides whether to draw a tile based on whether it falls within the camera's field of view. The key idea is that tiles are likely to be completely out of view. For instance, using the camera view that views the car from behind, the viewing frustum will contain only those tiles within a certain angle of where the camera is looking (the field of view).

However, this algorithm an ideal visibility detection algorithm would have to optimize the case when the camera rolls and pitches the field of view such that it encompasses all x-y values. This can be seen when the camera is very far away from the vehicle, as in the static camera we implemented. Hence, the simple 2D optimization can be somewhat complicated. We chose to ignore this case.

We simplified the viewing frustum by treating it as an infinite cone. Since all world coordinates are z-positive, we need only to project the cone down on to the x-y plane to determine tile visibility. This projection is an infinite triangle, which can be handled by triangle rasterization (as in problem set 5).

One of the challenges with implementing the visibility detection algorithm was figuring out the projection of the viewing frustum in world space. It would have been nice if we were able to do the projection and other matrix arithmetic entirely within GLUT, but GLUT provides little support for this. Thus, we reinvented the wheel and implemented our own matrix routines.

The real frustum projection must be calculated from the frustum eye, center, and up vectors as defined in GL. This involved finding the roll, pitch, and yaw of the ray between center and eye (roll is determined by up) and transforming the frustum in standard position by these angles. The resulting rays are then projected onto the x-y plane (via stripping the z dimension) to give us two or more rays. The extreme rays are then used for triangle scan conversion to determine which tiles will be drawn on the current cycle.

Physics Engine The physical car is modeled as a single rigid body supported by four spring-dampers. The unsprung mass of the wheels and tires is neglected, as well as their rotating masses, including the rotating mass of the engine and drive train. To couple the modeled car with the world geometry (allowing it to drive on arbitrary surfaces), a ray

is traced from each wheel position down into the world. The length of this ray becomes the amount of compression of the spring at that corner. This compression is then converted to an applied force normal to the plane on which the wheel is rolling. This force is combined with tire forces in the plane, such as the lateral force produced when the steering wheel is turned, braking and driving forces. Gravity and aerodynamic drag act on the car's center of mass for this simulation, so they do not result in any torque being applied to the car. However, the wheel forces do. These forces and torques are in turn applied to a set of first order ordinary differential equations describing the motion of a rigid body in space. These equations are integrated by a fourth order Runge Kutta numerical integrator to produce the car's trajectory.

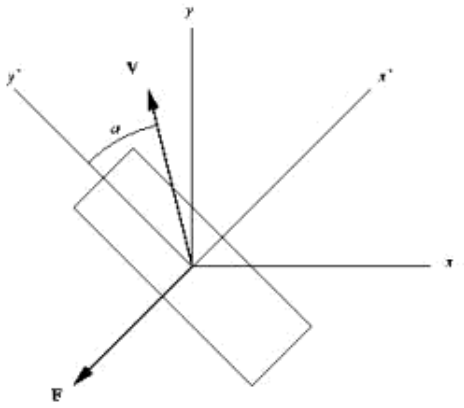


Figure 4.The calculation of tire lateral forces given by F = C x a, where a is the angle between the tire's rotated coordinate system and its velocity, and C is the tire's lateral stiffness.

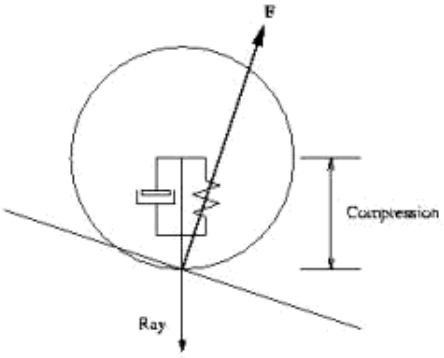


Figure 5. A diagram illustrating the normal force on a tire.

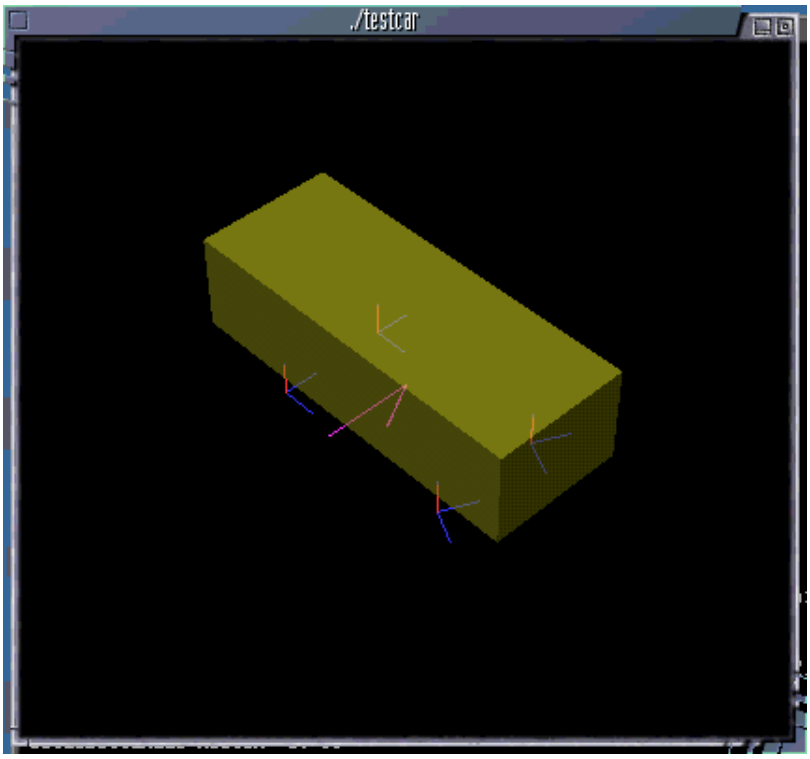


Figure 6. A snapshot of debugging output showing the tire coordinate systems (blue) and the normal forces on each wheel (red).

Collision Detection To implement collision testing, we performed ray-casting from the car and tested objects for intersection. Simplifying our code, we only implemented polygon intersection since our world is full of triangles. But instead of having the triangles stored in object root position (say in the x-y plane), the planes of the triangles are pre-computed and intersection occurs in "tile-space", the space defined by a tile centered at (100, 100) (before rotation and translation into the world grid). This saves computation and doesn't increase complexity.

Cameras Our game also includes a variety of different user-selectable camera angles. Our cameras provide the ability to watch the car from directly behind, the side, the top of the city, among

other perspectives. Figure 4 shows a side camera view, and Figure 5 shows a view of the car from the top of the city.

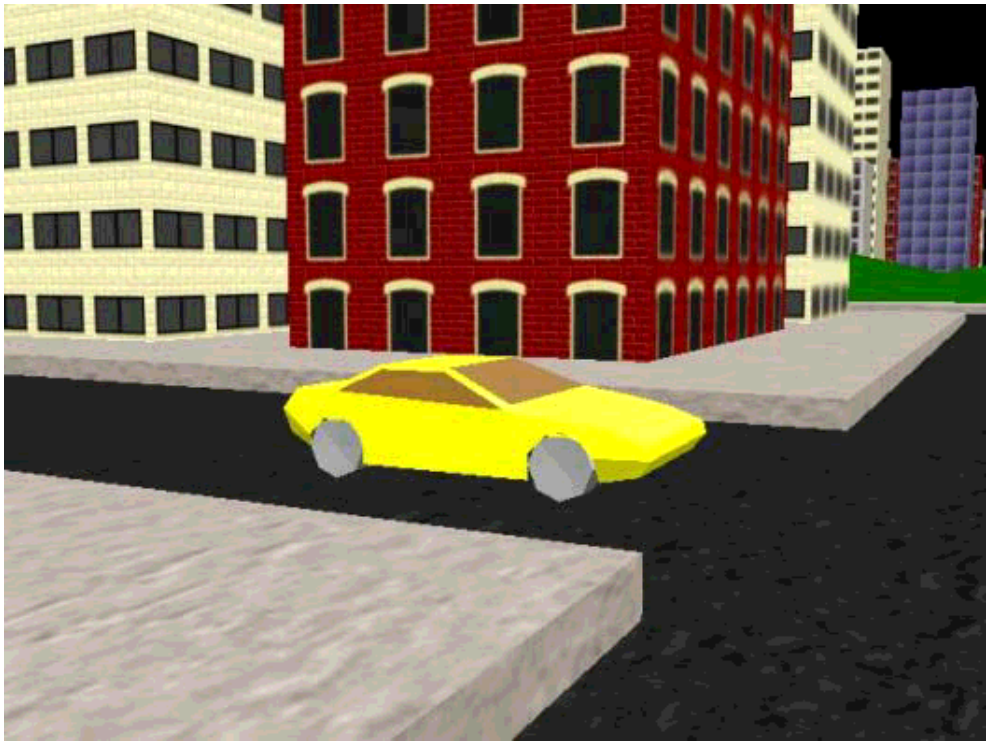


Figure 4. Screenshot of the city from the car's side.

Figure 5. Screenshot from the top of the city.

AI One of the difficulties for an AI in a driving game is how to make it stay on the road. Since cars only know where they are in the world in terms of coordinates, the AI would have to somehow be restricted from cutting through city blocks.

We originally had intended to solve this problem by generating waypoints, or fixed destinations, so that the AI could travel to any arbitrary point in the city, staying on the roads the whole time. However, to save time, we simplified the problem and created an AI that only drives around a city block. The AI works by accelerating towards the next corner of the block. Once it reaches its destination, it makes a hard left and holds the steering angle until the car approaches the desired angle. Once it does, the AI then makes minor corrections to the car's steering. It then sets its destination to the next corner and repeats the process.

Individual Contributions

Chad wrote all the matrix routines (which were used everywhere) and created the physics engine. Matt

modeled the city blocks, implemented the dynamic world loader, generated the textures, and added texture mapping to the city structures. Jordan wrote the visibility and collision detection routines, and created the user interface and some of the cameras. Stanley established the build environment, modeled the cars, wrote the AI, and created additional cameras.

We each fixed many bugs, improved various parts of the code, and on occasion, stepped on each other's work in the process.

Lessons Learned

Graphics Engine The parallel-file tile format is somewhat of a kludge. The most elegant solution would have been to write a parser for MAX files, which would have eliminated the need for the MAX to 3DS to RAW/RTX conversion stages. Also, we would have been able to take advantage of the extremely powerful texture coordinate mapping facilities in MAX. The RAW file parser was chosen simply because it was trivial to implement, allowing us to get real data into the application at an early stage of development.

Visibility Detection One lesson we drew from this work is that graphics optimizations are great, but they are generally difficult. Thank goodness they can also generally be simplified or hacked. Cones are nicer than frustums, but 2D cones (triangles) are nicer still. Due to time constraints, we couldn't go for a great, general visibility technique, such as BSP trees, but at least we can appreciate the work that need be done.

Making a game takes a lot of time We found that it took a long time just to get all the graphics components working. We lacked the time to implement everything that we had envisioned to make this a fully functional game.

OpenGL rocks We all learned to use OpenGL in a short time span and saw how easy it was to do many cool things with just a few lines of code. Without a standard graphics API that worked across many platforms, this project would have taken much longer.

Portability OpenGL and GLUT are wonderful libraries for making programs platform-independent, but we found that minor incompatibilities in C++ compilers caused problems from time to time. For instance, g++ requires that non-template friend functions include a "<>" after the prototype definition, while Visual C++ complains if those symbols are there.

Getting the program to work across many different platforms was a boon for development, however. With so many O2 systems taken by other students, we had the flexibility of using the Sun Ultras or working at home.

Acknowledgements

We would like to thank Damian Isla for his guidance and the patience to sit through our meetings, which often degenerated into off-topic discussions of crazy and wild things we could add to the game.

Bibliography

Huag, Edward J (1989). Computer-aided Kinematics and Dynamics of Mechanical Systems. Massachusetts: Allyn and Bacon.

International Federation of Automobile Engineers' and Technicians' Associations (1992). Total vehicle dynamics. London, England: International Congress.

OpenGL Architecture Review Board (1999). OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2 (Third Edition). Massachusetts: Addison-Wesley.

Appendix

To compile the project, load the source files into a directory and type "gmake". GL must be installed in the system, and the GLUT libraries and header files must be present in the directory "..\platform", where "platform" is either "irix", "solaris", or "linux".

For Windows machines, load the "final.dsp" file into Visual C++ and select "Rebuild All" from the "Build" menu. The PATH environment variable should contain a pointer to the GLUT dll, and the "..\win32" directory should be include the GLUT libraries and header files.

Command-line options:

-help output an extensive help message and quit

-cityfile %s city file (default: city.gnf)

-minfilter %s texture minification filter (GL_NEAREST, GL_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR)

-magfilter %s texture magnification filter (GL_NEAREST, GL_LINEAR)

-quality %s rendering quality (bad, medium, good, excellent)

-model %s user car model (yellow, red, limo)

-debug set debug flag

---

[1] Any references to 6.837 staff are purely for entertainment purposes only. The members of this group take no responsibility for any offense generated by the artwork. No animals were hurt during the creation of this game.