Making Next-Gen Video Games in Your Basement

Korhan Bircan
Department of Computer Science
Brown University
Providence, RI 02912
kbircan@cs.brown.edu

Abstract

The combination of open source libraries with the advanced and affordable hardware has made near commercial quality video game production a reality for thousands of hobbyists all around the world. Never before have the graphics looked so good nor the physics felt so real with such high frame-rates. In this project we have demonstrated how decent quality video games can be made without an army of programmers or millions of dollars investment. Even though the techniques that we have implemented are proof of concept, the results are promising and with more time and effort and more extensive use of the technologies we will describe shortly, they can be improved to a level where the line between commercial quality and homemade games will become blurry.

One of our aims was to create a code base for the enthusiastic prospective students of CS134 Innovating Game Development class. We have compiled a set of tutorials for some of the most practical freely available libraries and engines. We hope that our work will provide the students with a set of tricks and techniques for rapid game prototyping and guide them in the right direction so that the transition from hobbyist game programming to professional game programming will be a seamless one (if they choose to go in the industry in the future).

Keywords: Game development, XNA, Irrlicht, Ogre 3D, Newton Game Dynamics, .ODE, NVIDIA PhysX, OpenAL, FMOD, ragdoll physics, vehicle simulation, particle systems, projective texturing, high dynamic range rendering, Wiimote control, speech recognition.

1 Introduction

Our main aim in this project was to determine the best set of tools for CS134 students in order to enable them to prototype and create games quickly and also demonstrate the power of these tools with a proof of concept game. We went through some of the most popular rendering, physics, and sound engines before deciding which libraries we should use for this project. We have prepared a set of tutorials for each of the engines and libraries that we examined. Here is a brief explanation of some of the technologies we have looked at.

2 Practical Tools for Making Video Games on Your Own

PC gaming has taken big hits in the last couple of years. During 2007, the sale of PC games generated only 14% of the gaming industry's $18.85 billion software revenue, according to figures released from market research firms[1]. This must have rung alarm bells at semiconductor companies such as Intel, AMD, and NVIDIA whose revenues are greatly affected by PC gamers. In order to promote PC game development, Intel has acquired Havok Inc[2]., one of the leading providers in physics technology for gaming and digital content, and NVIDIA has acquired AGEIA Technologies[3], who is another leading provider in gaming physics technologies. Microsoft has been promoting PC game

[1] http://www.shacknews.com/onearticle.x/50939
[2] http://www.intel.com/pressroom/archive/releases/20070914corp.htm
[3] http://www.nvidia.com/object/io_1202161567170.html

development in various ways, and has developed a software development framework, XNA, in 2004 and recently has made it even more attractive by enabling developers to deploy their games on XBox 360 and also let them share user created games via XBox Live. There are already some very stable open source rendering, physics, and audio engines. The Internet is an ocean of game development related information. In short, game development has never been so accessible to the mass public. The hardware has never been so good at rendering such realism, physics simulations have never run so fast (now some even run on the GPU eg. PhysX) and accurate, shaders have never been so powerful, and there has never been such a plethora of intuitive tools and API's. With such a variety to choose from, it could be a little confusing for a beginner to figure out where to start. We went through the most popular and stable freely available engines and libraries out there and compiled our findings as a set of tutorials which can be downloaded from the author's webpage[4].

2.1 XNA

Microsoft XNA Game Studio Express is a set of tools based on Microsoft Visual Studio C# that allows hobbyists to build games for both Microsoft Windows and Xbox 360. XNA uses DirectX API to render the graphics. DirectX is a relatively complicated API compared to OpenGL. It takes much longer to learn DirectX, system initializations and setup are fairly tedious. XNA is a tremendous leap in that regard. It makes using DirectX almost a breeze. Displaying 3D models, getting user input, math operations, storage, playing sounds and other media files are seamless thanks to the strong DirectX media framework. Integrating HLSL shaders have never been so easy in any other framework. Just that aspect itself carries so much importance that it may be enough reason to use XNA. The exact same code written for the Windows operating system can be built and run on Xbox 360 through the XNA framework. There were two reasons why we did not choose XNA for our project. XNA is not a game engine. It is a game development framework. Therefore primitive things such as adding a camera to the scene, making it first person style, third person style, animations, particle effects, template shader effects, template models, and media are missing. After one implements those missing pieces, then XNA would be a great prototyping tool. The second reason is to do with runtime performance. The C# compiler is at present not as sophisticated as the Visual C++ compiler and it may end up creating performance problems[5]. We have compiled an introduction tutorial to get started with XNA[6].

2.2 Irrlicht

Irrlicht is an open source cross-platform realtime 3D engine that uses DirectX, OpenGL, and its own software renderer. Some of its main features are extensible material library with vertex and pixel shader support, customizable scene management, character animation system through skeletal and morph target animations, particle effects, billboards, light maps, environment mapping, stencil buffer shadows, water surfaces, bump and parallax mapping, sphere mapping, texture animation, 2D GUI system, 2D drawing, direct import of textures and common mesh formats, collision detection and response system, optimized 3D math library, and integrated XML parser. As with most popular

[4] www.cs.brown.edu/people/kbircan

[5] An XNA program manager and a CS134 guest speaker Frank Savage, gave as an example a performance problem where the C# compiler did not inline an overloaded operator but instead made function calls. Detailed explanation of this problem was addressed in the following technical article "Optimization Tutorial: Particles and High-Frequency Code", http://creators.xna.com/Headlines/tutorialscol1/archive/2007/08/09/Optimization-Tutorial_3A00_--Particles-and-High_2D00_Frequency-Code.aspx

[6] Useful Tools for Making Games: XNA, http://www.cs.brown.edu/people/kbircan/talks/xna.ppt

open source projects, one of its strength is the large user community. The forums and user created tutorials and games are valuable resources. It is fairly simple to make simple games with Irrlicht in a couple of hundred lines of code. We have prepared an introductory tutorial which can be found at the provided link[7].

2.3 Ogre

Ogre (Object-Oriented Graphics Rendering Engine) is the another cross-platform 3D engine that abstracts the details of using the underlying system libraries such as Direct3D and OpenGL and provides an interface based on world objects and other intuitive classes. It has more features than any of the engines and frameworks we have looked at. It is relatively large and the learning curve is a little steeper. The way the engine was designed is reminiscent of the Quake engine. The materials are declared in .material script files, particle systems are declared in .particle script files, shader parameters are tied to parameters in your program in .program files. The 3D models need to be converted into Ogre's proprietary .mesh format. These are some of the things that make the learning curve a little steeper but once comfortable, this is as good a graphics engine as most commercial ones. It comes with a very impressive set of tutorials, materials, and shaders which make it all the more inspirational to start making a game. Samples out of the box include Bezier patches, BSP level loading, BSP collision detection, camera animation, cel shading, various shader effects, crowd simulation, cube mapping, deferred shading, bump mapping, parallax mapping, environment mapping, facial animation, Fresnel refractions, grass simulation, 2D GUIs, instancing, dynamic lighting, ocean simulation, particle effects, rendering to textures, texture animation, dynamic shadows, skeletal animation, sky boxes, sky domes, sky planes, smoke rendering, terrain rendering, transparency, volumetric textures, and water surface simulation. What is more is the biggest online user community. It has the largest forum database and wiki entries, user created tutorials and other content. Suffice it to say we have chosen Ogre as our rendering engine. We have provided an introductory tutorial to help students get started using the engine which can be downloaded at the provided link[8].

2.4 ODE

Open Dynamics Engine (ODE) is an open source library for simulating rigid body dynamics. It emphasizes speed and stability over physical accuracy. It uses special non-penetration constraints when two bodies collide. The alternative method used by some other physics engines is virtual springs which are error-prone. It supports rigid bodies with arbitrary mass distribution, various joint types (ball, hinge, slider, fixed, angular motor, and universal), a number of collision primitives (sphere, box, capped cylinder, plane, ray, and triangular mesh), quat-tree, hash-space, and simple collision spaces. It derives its equations of motion from a Lagrange multiplier[9] velocity based model and uses a first order integrator. The available time stepping methods are either standard big matrix or an iterative method. The contact and friction model is bassed on the Dantzip LCP solver described by Baraff[10].

2.5 NVIDIA PhysX

When we started this project the company was owned by Ageia and one had to jump through hoops to get the SDK. Recently NVIDIA has bought PhysX and made it freely available and ported it to work on its GeForce 7900 and higher series GPUs. The engine supports complex rigid bodies, a number of collision

[7] Useful Tools for Making Games: Irrlicht, http://www.cs.brown.edu/people/kbircan/talks/irrlicht.ppt

[8] http://www.cs.brown.edu/people/kbircan/talks/ogre.ppt

[9] In mathematical optimization problems, Lagrange multipliers are used to find the extrema of a function of several variables subject to one or more constraints.

[10] "Physically Based Modeling: Principles and Practice", SIGGRAPH '95 course, David Baraff.

primitives including sphere, box, capsule, plane, heightfield, convex shapes, and triangular meshes, includes the widest array of joint types such as spherical, revolute, prismatic, cylinders, fixed, distance, pulley, 6DOF, has a very advanced ragdoll creation and editing system, supports materials and friction model, continuous collision detection (CCD). A great feature is the advanced character control system which can be used for first-person or third-person player control that does not make use of rigid body physics. It has a sophisticated articulated vehicle dynamics which supports various wheel shapes and joint-based suspension. It is able to simulate volumetric fluids and gases using particle systems and emitters. Fluid simulation mode can be either Smoothed Particle Hydrodynamics (SPH) or simple mode without inter-particle forces. The volumetric force field simulator supports various force field shapes such as sphere, capsule, box, and convex mesh and allow the simulation of effects such gust of wind, dust devils, vacuum cleaners or anti-gravity zones. Another distinct feature of the engine is the cloth and clothing authoring and playback which supports cloth attachment, self-collisions, tearing, pressure, and deformable metal cloths. The last but not the least it allows simulation of volumetric deformable objects and provides a tool called NVIDIA TetraMaker for easy soft body creation.

2.6 Newton

Newton Game Dynamics is an integrated solution for real time simulation of physics environments. The API is small, stable, and provides scene management, collision detection, and dynamic behaviors. It is unfortunately closed source. In contrast to most other real-time physics engines it favors accuracy over speed. It has a deterministic solver not LCP[11] or iterative methods. Some of its main features are having a wide variety of collision shapes, continuous collision mode, and hinge, ball, slider, corkscrew, and custom joints. We have chosen to use Newton in our project because of the easy integration, excellent documentation, wide range of samples, and the large user community. Another factor is the stable and relatively easy convex hull support which we used extensively for some of our 3D models. We have prepared an introductory tutorial to help get started with Newton which can be downloaded at the provided link below[12].

2.7 OpenAL

OpenAL (Open Audio Library) is a cross-platform (Windows, linux, and Macintosh, iPhone, and Xbox 360) library that models a collection of audio sources moving in a 3D space that are heard by a single listener somewhere in that space. The basic OpenAL objects are a Listener, a Source, and a Buffer. There can be a large number of Buffers, which contain audio data. Each buffer can be attached to one or more Sources, which represent points in 3D space which are emitting audio. The result of this is that the sounds behave naturally as the user moves through the 3D environment and the programmer does not need to perform much additional work.

2.8 FMOD

FMOD is a closed-source sound engine that supports more hardware platforms than any other audio system including GameCube, Wii, PlayStation Portable, Playstation 2, Playstation 3, Xbox, Xbox 360 as well as various operating systems including Windows, Linux, and Macintosh. Among its key features are digital sound processing effect suite, 2D/3D morphing of sound, and 3D reverb support and geometric occlusion. One

[11] In linear algebra, linear complementarity problem (LCP) consists of starting with a known n-dimensional column vector q and a known nxn matrix M, and finding two n-dimensional vectors w and z such that:

- q = w - Mz
- w_i >= 0 and z_i >= 0 for all i
- w_i x z_i = 0 for all i

[12] http://www.cs.brown.edu/people/kbircan/talks/newton.ppt

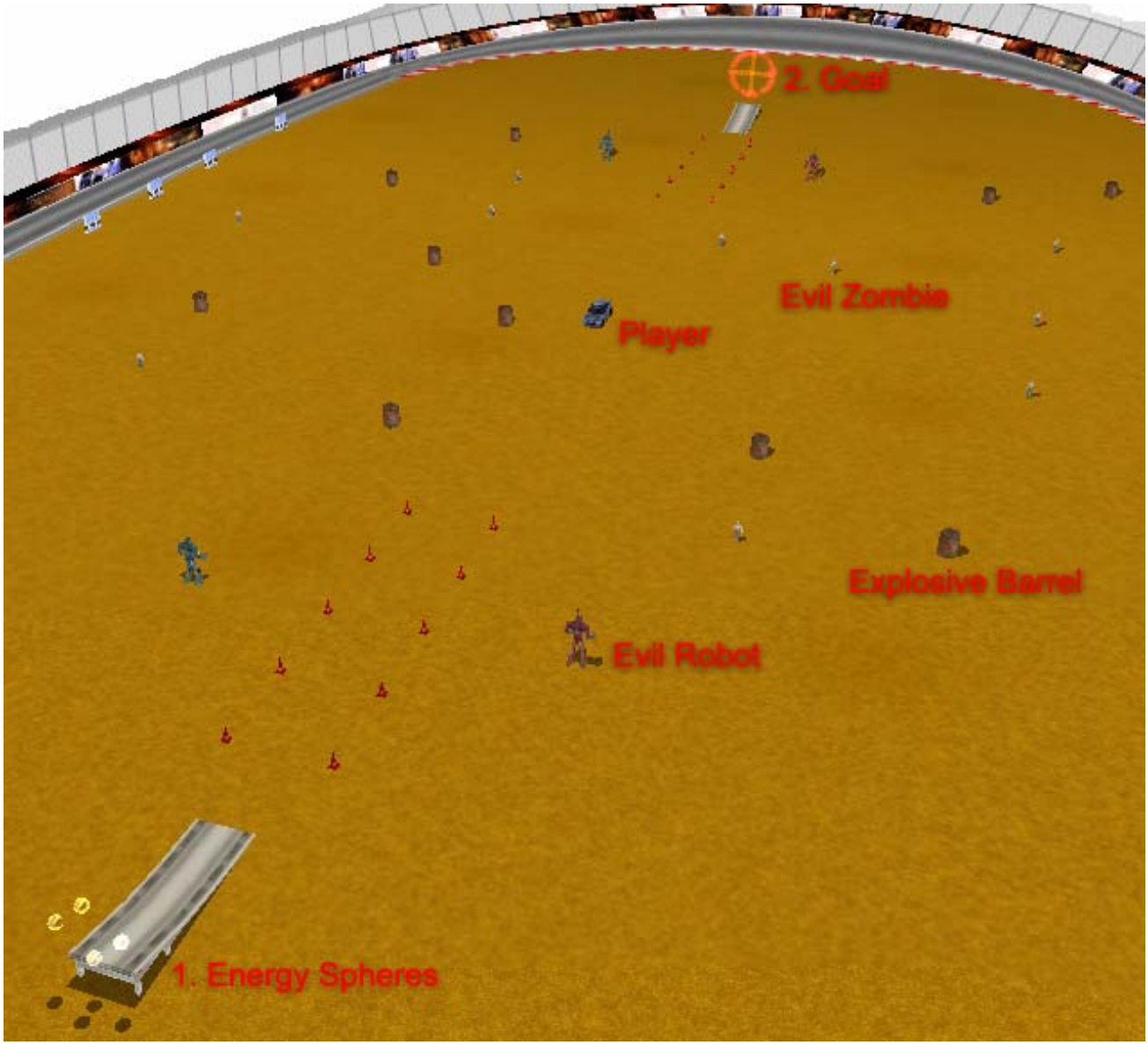

Figure1. A bird's eye view of Twisted Race..

impressive feature is a tool called FMOD Designer which allows simple or complex multilayer sound events to be modeled and

created by the sound designer. This way the programmer is provided with assets and an event list and the behavior of the audio events are independent of the game implementation. We have prepared an introductory tutorial to help get started with Newton which can be downloaded at the provided link below[13].

[13] http://www.cs.brown.edu/people/kbircan/talks/fmod.ppt

3 Twisted Race the Game

The game we created as part of this project is a fair example of what can be done in terms of rapid prototyping using a combination of these engines and libraries. We hope that it can be used by future CS134 students as a template implementation for some of the fundamental next-generation video game tricks as well as a reference for standard requirements of any game (graphical user interfaces, game state handling, AI, game control loop, player control, user input handling etc.). We also believe that our code base will be a valuable resource for CS134 students who want to learn the theory and practice behind some

fundamental game programming techniques. In this regard we have demonstrated how techniques such as ragdoll physics, vehicle simulation, particle effects, and various shader effects can be implemented and integrated into a game. As far as innovative user interaction goes, we have demonstrated how Wiimote controls and speech recognition can be a natural and fun part of game dynamics.

The goal of the game is to collect the energy spheres (shown as number 1 on the figure below) and place them on the ring at the other side of the map (location number 2 on the map). The player is a race car driver and controls a realistically simulated rally car. In order to collect a sphere, the player has to fly over a ramp at one end of the map. To place a sphere, the player has to fly over the other ramp and through an energy ring at the other end of the map. Both ramps are protected by evil robots who shoot missiles at the player once close enough. The player can dodge those missiles or blow these robot up by shooting missiles himself. There are evil zombies lurking around who want to stop the player from driving around. These zombies are immune to fire power and can only be defeated by physical impact. To make things harder, there are explosive barrels placed randomly in the map, which the player can either dodge or blow up to get more points. The sooner the mission is completed and the more enemies and obstacles destructed, the higher the score. Players take turns trying to finish each level in the shortest amount of time. With each new level the enemies get more aggressive and there are more obstacles in the way.

We have chosen to use Ogre as the rendering engine due to the largest user community, user created contents, and commercial quality rendering capabilities. We are using Newton as our physics engine for easy integration but as soon as NVIDIA made PhysX SDK freely available Newton no longer was the optimal choice, however it was too late to change our infrastructure. We have chosen to use FMOD for sounds because of the wide commercial use in console games. Not having to change code across any platform is a huge benefit and the engine has been used in triple A titles such as Guitar Hero III, BioShock, Heavenly Sword, Call of Duty 4, World of Warcraft, and StarCraft II therefore it was obvious that we would endorse this engine.

3.1 Ragdoll Physics for Zombie Simulation

Ragdolls are physics simulated game characters where the bones are represented as rigid bodies and are tied to each other via joints. Ragdolls are usually used as a replacement for traditional static death animations. There are various implementation approaches to ragdoll physics. Verlet integration[14] models each character bone as a point connected to an arbitrary number of other points via simple constraints. Inverse kinematics[15] post-processing relies on playing a cooked death animation and then using inverse kinematics to force the character into a possible position after the animation has completed. In procedural animation is used in non-realtime media and employs use of multi-layered physical models (bones, muscles, nervous system) in non-playable characters (NPCs). The qualities of the movement are more natural and provide a more immersive experience. In blended ragdoll a cooked animation is used as a constraint to the output of the physical system would allow. A state of the art implementation of this technique can be found in a dynamic motion synthesis software package called Endorphin by NaturalMotion[16] which uses behavior scripts to combine physics, AI, and genetic algorithms to create realistic animations for game characters.

[14] Method for calculating the trajectories of particles in molecular dynamics simulations.
[15] Process of determining the parameters of a jointed flexible object in order to achieve a desired pose.
[16] NaturalMotion, http://www.naturalmotion.com/

We have chosen to use a constrained rigid body approach where we represented each bone as capsules and connected them with ball-and-socket or hinge joints. Here we may briefly talk about the various rigid body and joint types. Rigid bodies can be one of primitive box, capsule, or filled-donut shaped chamfer cylinder. They have properties such as size, orientation, position, mass, inertia matrix, linear and angular velocities. Joints define the type of constraints between two connected rigid bodies. Ball and socket joint constraints the ball of one body to be in the same location as the socket of another body. Hinge joint constraints the two parts of the hinge to be in the same location and lined up along the hinge axle. Slider joints have a piston and socket and they constraint the two rigid bodies to line up along these two.

We have developed a scalable system that can be used to create ragdolls for any 3D model with a skeleton. The rigid bodies and joints need to be defined in an XML file in the format shown in figure1. The ragdoll physics simulation is turned on at the moment of impact. Till then the model is associated with a capsule shaped bounding volume. The graphics scene node plays the animation and the character movement is governed through applying forces to the bounding volume. The bounding volume is assigned custom contact callback functions for desired materials in the scene (eg. other characters, vehicles, missiles, props etc.). Through the callback function the bounding volume is deleted, the animation is stopped, and the ragdoll simulation is started. From there on, the rigid bodies behave according to the forces due to the impact, gravity, friction, and the constraints created by the joints. We let the ragdoll simulation run for a certain period of time and delete the ragdoll bodies after the system has reached equilibrium. We have observed that getting rid of the ragdoll bodies improves the performance of our global simulator.

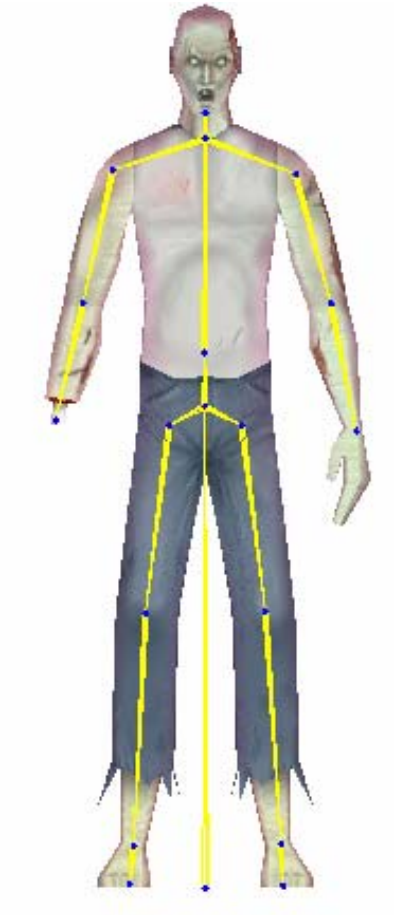

Figure2. Skeleton we use for the 3D character model,

```
<RagDoll>
  <Bone dir="0 1 0" length="0.0"
   shape="hull" size="0.1 0.0 0.0"
   mass="10.0" skeleton_bone="NECK">
    <Joint type="ballsocket"
     pin="0 1 0" limit1="20.0"
     limit2="10.0" />
    <Bone dir="0 1 0" length="0.0"
     shape="hull" size="0.5
     0.0 0.0" mass="8.0"
     skeleton_bone="HEAD">
      <Joint type="ballsocket"
       pin="0 1 0" limit1="40.0"
       limit2="30.0" />
    </Bone>
  </Bone>
  ...
</RagDoll>
```

Figure3. XML based rigid body and joint input we provide to our system.

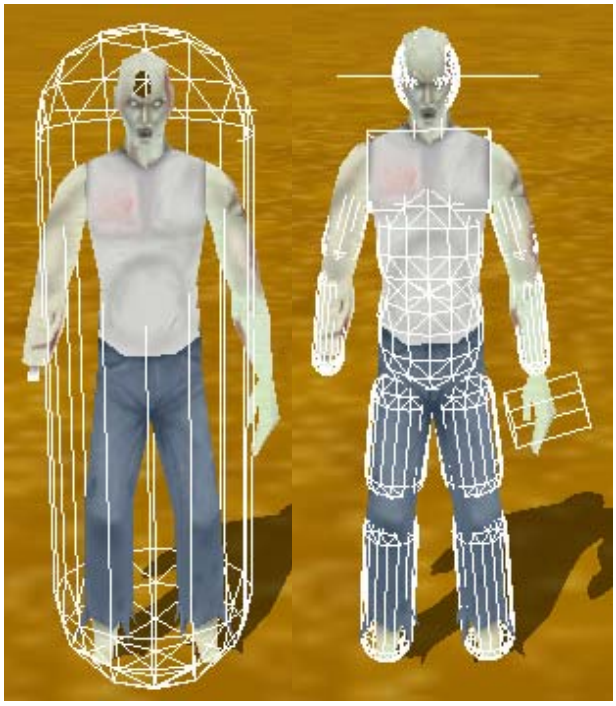

Figure4. On the left is the bounding box view of the character and on the right is when the ragdoll is actually turned on.

Figure5. Ragdoll system after having reached a stable state.

3.2 Vehicle Simulation for the Playable Character

We have simulated a relatively realistic vehicle by attaching four tires to a rectangular prism chassis via ball and socket joints. The tires have properties such as local position/orientation, pin position/orientation on the chassis , mass, width, radius, spring damper coefficient, suspension spring constant, and suspension length. The vehicle is driven by first setting the steering angle then setting a desired torque to the front tires. Spring and suspension related properties help simulate a variety of vehicles. Adjusting the friction coefficient between the tires and the various regions of the world (eg. dirt ground vs asphalt track) help convey the different driving conditions. A nice extension would be to have different weather conditions such as rain and snow and simulate the traction by adjusting the friction coefficients.

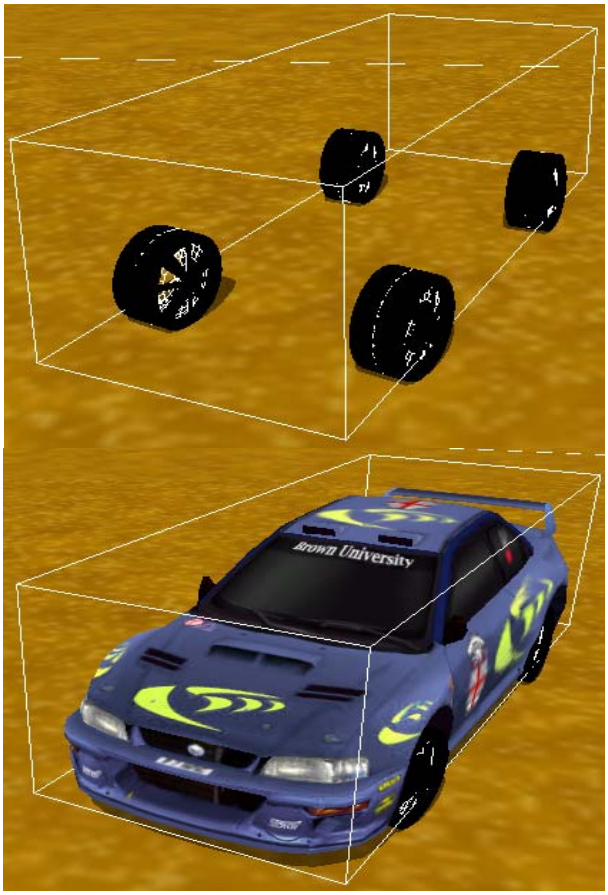

Figure6. Our vehicle system. On the left the chassis and the four wheels, on the right a 3D model is attached to the physics system.

3.3 Particle Effects for Fire, Smoke, and Dust

We have come up with creative ways of using particle effects in a number of occasions. There are three major properties of particle systems. These are particle system attributes such as the maximum limit of allowed particles, the texture file, dimension of the texture, and billboard type; emitter attributes such as the type of emitter (box, cylinder, ellipsoid, point, ring), its position , velocity, color, and time-to-live. The last property of our particle systems are called affectors and they can be used to apply linear forces, fade particle colors, scale, rotate, or apply custom transformations to the particles. Here is a sample particle system script that we came up with that creates an explosion effect:

```
explosion
{
quota           200
material        explosion
particle_width  6
particle_height 6
cull_each       false
renderer        billboard
billboard_type  point

emitter Point
{
      angle 101.7
      colour 1 1 1 1
      colour_range_start 1 1 1 1
      colour_range_end  1 1 1 1
      direction 0 1 0
      emission_rate 200
      position 0 0 0
      velocity 5
      velocity_min 4
      velocity_max 4
      time_to_live 0.2
      time_to_live_min 0.5
      time_to_live_max 0.5
      duration 0.2
      duration_min 0.5
      duration_max 0.5
      repeat_delay 0
      repeat_delay_min 0
      repeat_delay_max 0      }

affector ColourFader
{
      red 0.9024
      green -0.3913
      blue -0.1
      alpha 0.7073
}
}
```

Figure7. A sample execution of our particle system.

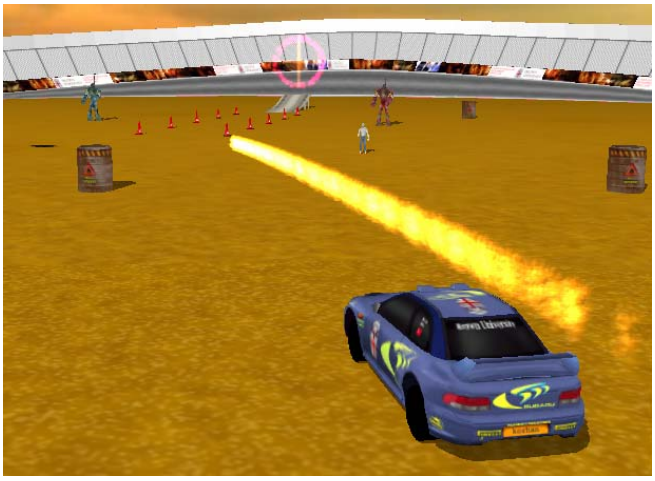

Figure8. A variation of the explosion script used as rocket smoke.

Figure9. Dust particles when a vehicle skids.

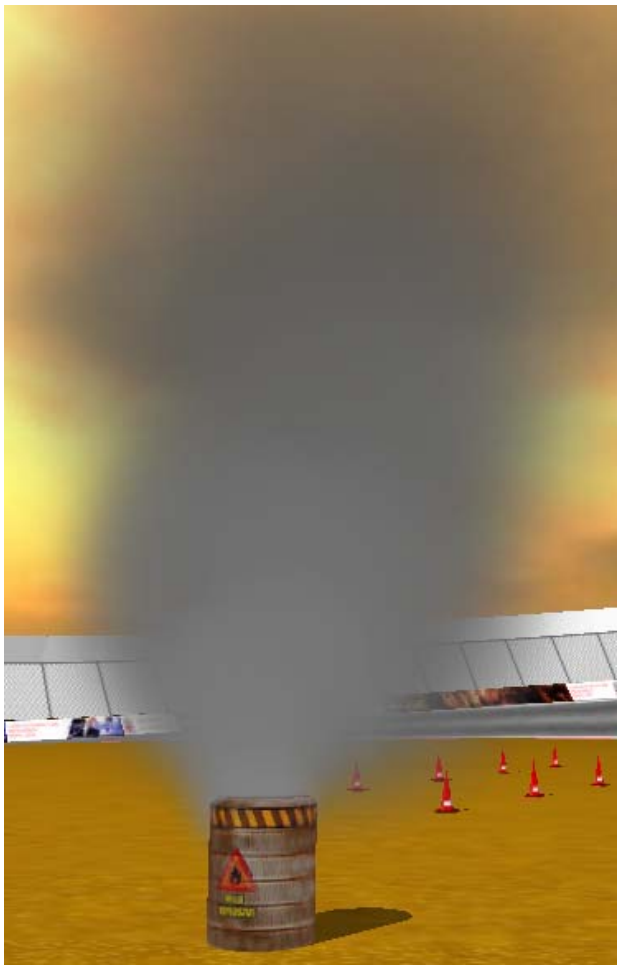

Figure11. Smoke particles after a barrel is hit.

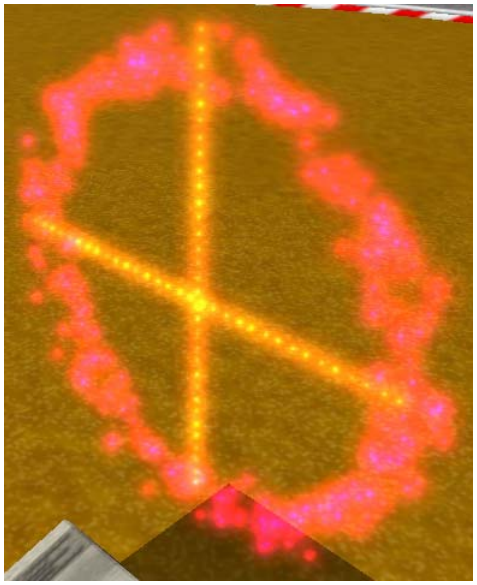

Figure12. A ring and two linear particle systems (similar particle systems are used to create laser beam effects in other games).

We have made all the scripts and materials to these particle systems available under the Media/particle/ and Media/materials/textures/ directories respectively.

3.4 Projective Texturing for Craters and Blood Pools

Projective texturing allows a textured image to be projected onto a scene as if by a slide projector. It is essentially a special per-vertex matrix transformation which is interpolated linearly in the pixel shader.

1. Create decal frustum (set near/far clip planes, position, and orientation)
2. Get the material
3. Create a new pass in the material to render the decal
4. Set our pass to blend the decal over the model's regular texture
5. Set the decal to be self illuminated instead of lit by scene lighting
6. Setup decal texture unit
7. Enable projective texturing, set texture addressing mode and texture filtering properties

How the shaders achieve step 6 can be summarized as the following:
7.1 Render the scene from the light's point of view
7.2 Use the light's depth buffer as a texture
7.3 Protectively texture the shadow map onto the scene
7.4 Use texture colors in fragment shader

As a result of this procedure desired textures can be projected onto non-flat surfaces as well as any object in the scene. The following figures show in-game examples of how we used this technique.

This is a common technique used in almost all next-gen first person shooter games to place

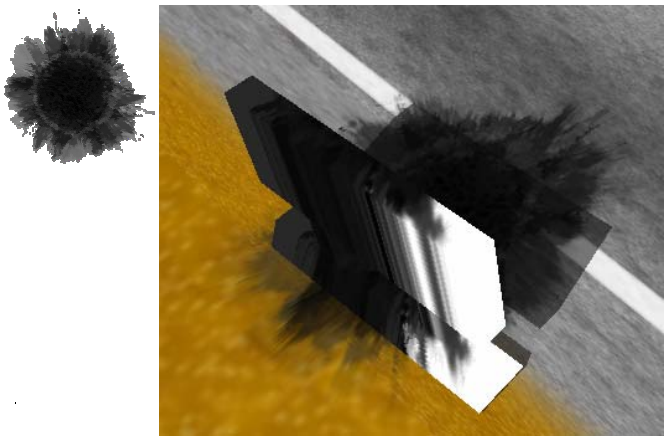

Figure13. Crater texture projected on an arbitrary object in the scene.

Figure14. The same crater texture from Figure6. rotated and scaled randomly to give the impression of non-static appearance.

decals on walls where bullets hit. An obvious improvement to this technique is to use bump-mapped textures so that the crater or the gun shot decal looks carved into the surface.

One important issue we have observed is that with individual pass of projective textures there is significant loss in frame rates due to additional shader passes. We have witnessed a possible driver bug (ATI Catalyst April 16, 2008 for ATI X600, DirectX 9.0c 4.09.0000.0904) where all textures in the scene go gray after the 16th decal is projected on the scene. To circumvent this issue we have decided to preallocate our decals in a queue of size 10, then recycle the decals in a first-in-first-out fashion. This technique also proved to be faster because no dynamic allocations/deallocations were being made. Another common approach is to delete the decals after a certain period of time which has the drawback of dynamic memory allocations/deallocations.

3.5 High Dynamic Range Rendering (HDR)

HDR lighting is a technique to render highly realistic lighting effects by using floating-point textures and high-intensity lights. Unlike the traditional textures in integer format, floating-point textures are capable of storing a wide range of color values. Because color values in floating-point textures don't get clamped to [0, 1] range, much like lights in real-world, these textures can be used to achieve greater realism.

Figure15. A shot of the car in front of the sky before and after bloom shader has been applied.

Image processing is usually done offline on the CPU. With pixel shaders they can be efficiently performed on the GPU as a post-processing effect allowing them to be applied in real-time. In this project we have used HDR to create bloom which is a widely used next-gen shader effect. The visual effect that bloom achieves is that bright spots in a scene get magnified and the area around these spots are further brightened. In order to achieve this, two passes of down filter are first performed. This can be done because the bright spots do not need to be very detailed. Next, a bright pass is done where the lower intensity regions

of the image are stenciled out. After that, the current image is blurred out once in the horizontal and vertical directions. As a final step, two passes of up filter are performed to bring the image back to original size and then the original image and the processed image are combined.

3.6 Wiimote Control

Since the Wiimote was announced in September 2005 it has received much attention due to its motion sensing capability, which allows user interaction through the use of accelerometer[17] and optical sensor technology. There is an open source movement called Wii homebrew which provides tools to expand or alter the capabilities to reuse the Wiimote hardware, accessories, and software. For our game we have decided to use the accelerometer of the Wiimote to control the steering, forward and backward acceleration of the vehicle. We have used the Wii Remote Communication API for which allows to connect the Wiimote through Windows' Human Interface Device (HID). This set of classes support receiving button presses and motion data as well as setting rumble and LED status. We connect to the Wiimote through Windows' Bluetooth HID service. We have used the BlueSoleil Bluetooth Stack[18] and TrendNet Bluetooth adapter[19].

Here is a simplified outline of the steps taken to communicate with the Wiimote:

1. Start listening to input reports
2. Request the Wiimote to start sending back motion data
3. while (game_is_running)
    3.1 Get last motion data
    3.2 Interpret motion data
4. Stop listening
5. Disconnect

The motion data needs to be averaged over the last several inputs to smooth out the players motion. Also, when the Wiimote is pitched, gravity increases the motion value readings which causes problems when the Wiimote is shook, therefore initial calibration is required and the inputs need to be normalized and clamped to achieve a smooth game play experience.

3.7 Speech Recognition

We have integrated speech recognition as a part of aiming system for the vehicle. The player can use one of the commands {"left", "right', "up", "down", "stop", "fire", "reset"} to control to cross hair and shoot a missile. We have used Microsoft Speech SDK SAPI 5.1[20] for speech recognition (SR) capability. SAPI is a very intuitive interface which enabled us to initialize the SR engine, accept input from the microphone, and recognize words in a few lines of code.

Before speech recognition takes place, the SR engine requires grammar creation and grammar activation. A recognizer may have more than one grammar associated with it which can be of type context free grammar[21] or dictation. We load the general dictation topic file and activate it, after setting the audio options and dictation states, we launch a new thread that blocks for audio input. We have

[17] A device for measuring acceleration and gravity induced reaction forces. Singe- and multi-axis versions models are available to detect magnitude and direction of the acceleration as a vector quantity. They can be used to sense inclination, vibration, and shock.

[18] http://www.bluesoleil.com/

[19] http://www.trendnet.com/products/proddetail.asp?prod=130_TBW-102UB&cat=10

[20] http://www.microsoft.com/downloads/details.aspx?FamilyID=5e86ec97-40a7-453f-b0ee-6583171b4530&displaylang=en

[21] A context-free grammar (CFG) consists of a number of productions. Each production has an abstract symbol called a nonterminal as its left-hand side, and a sequence of one or more nonterminal and terminal symbols as its right-hand side. For each grammar, the terminal symbols are drawn from a specified alphabet. In this context a CFG refers to a user created set of rules that would constitute a grammar (eg. If we were developing an application to accept orders via phone, we would create a grammar to define of acceptable user input.)

observed that recognition success rate is highest for single word commands. We have compiled a list of words that sound like the acceptable commands to smooth out the erroneous recognition (eg. “resend” ~ {“resends”, “resent”, “reset”, “resets”, “recent”} etc). The SR engine does a fairly good job of recognizing player commands in real-time as long as there is not too much noise in the environment.

4 Future Work and Conclusion

We believe that our project constitutes a sufficient reference code base for most genres of games that can be developed within a semester. Apart from the innovative aspects and the various techniques we have implemented, the project is a sample work of how rendering and physics should be integrated, GUIs and HUDs are added, in game sound should be used, game state and character controlling should be handled, and in general how the code should be structured so that the various modules of the game can best communicate with each other.

We are very excited with the acquisitions of Havok by Intel and Physx by NVIDIA and we are looking forward to getting our hands on the next versions of their SDK. Our project is not only is a fun game to play but also a playground of various technique implementations. We are eager to port the game to a more advanced physics engine and come up with new ideas so that we can integrate the interesting features these advanced engines have to offer (eg. fluid simulation, cloth simulation, soft body simulation, advanced character control etc.).

We think that our suggestions about what public engines and libraries to use will prove practical by saving CS134 students considerable amount of time in the beginning of the project. The tutorials we have provided still cover most of the technologies we have analyzed.

Game programming is a best suited for people who enjoy coding in their free times. It is unrealistic to expect any one course to cover all the available technologies and tools. Therefore the student’s curiosity and enthusiasm are the driving force to successful game programming. Game programming is probably the one area of software engineering that gets so much affected by hardware and technology trends. The new lines of CPUs, GPUs, video consoles, input devices, networking technologies, mobile devices, and many more immediately effects on video games. Unless the students are self-motivated and are successful in keeping up with the most recent technologies they will not be able to avoid falling behind

The game and gameplay videos/screenshots can be found at the author’s website[22].and the author will be more than happy to provide technical support with the game and related issues.

5 References

[1] Verlet Integration, http://en.wikipedia.org/wiki/Verlet_integration
[2] Ragdoll Physics, http://en.wikipedia.org/wiki/Ragdoll_physics
[3] Inverse Kinematics, http://en.wikipedia.org/wiki/Inverse_kinematics
[4] Projective Texture Mapping, Cass Everitt, Nvidia Developer Zone, http://developer.nvidia.com/object/Projective_Texture_Mapping.html
[5] DirectX and XNA Development Kit Documentation, March 2008.
[6] Wii Remote Communication API, http://digitalretrograde.com/projects/wiim/
[7] Wii Remote, http://en.wikipedia.org/wiki/Wiimote
[8] Wii homebrew, http://en.wikipedia.org/wiki/Wii_homebrew

[22] www.cs.brown.edu/people/kbircan

[9] The HID Page,
http://www.lvr.com/hidpage.htm
[10] Microsoft Visual Studio 2005 Express
Edition Documentation,
http://msdn.microsoft.com/en-
us/express/aa975050.aspx
[11] Irrlicht, http://irrlicht.sourceforge.net/
Ogre 3D, http://www.ogre3d.org/
[12] Newton Game Dynamics,
http://www.newtondynamics.com/
[13] Open Dynamics Engine, http://www.ode.org/
[14] NVIDIA PhysX,
http://developer.nvidia.com/object/physx.html
[15] OpenAL, http://www.openal.org/