Teaching Computer Game Design and Construction

Scott Schaefer, Joe Warren
Rice University*

Abstract

Computer gaming is a key component of the rapidly growing entertainment industry. While building computer games has typically been a commercial endeavor, we believe that designing and constructing a computer game is also a useful activity for educating students about geometric modeling and computer graphics. In particular, students are exposed to the practical issues surrounding topics such as geometric modeling, rendering, collision detection, character animation and graphical design. Moreover, building an advanced game provides students exposure to the real-world side of software engineering that they are typically shielded from in the standard computer class. In this paper, we describe our experiences with teaching a computer science class that focuses on designing and building the best game possible in the course of a semester. The paper breaks down a typical game into various components that are suited for individual student projects and discusses the use of modern graphical design tools such as *Maya* in building art for the game. We conclude with a rough timeline for developing a game during the course of a semester and review some of the lessons learned during the three years we have taught the class.

1 Overview

In the fall of 2000, one of the authors (Warren) returned from his sabbatical to teach the introductory computer graphics class at Rice University. This class followed the standard mold of most introductory graphics classes using the textbook by Foley et al. [Foley et al. 1990]. Normally, at Rice, this course is followed by a class in the Spring that focuses on a related advanced topic such as geometric design, visualization or computational geometry. In the spring of 2001, Warren decided to create a new class whose sole focus was based on the following goal: *As a class, build the best computer game possible in the course of a single semester*. After its third incarnation here at Rice (the last in spring 2003), this class has proven to be a remarkably successful tool for motivating students to learn about 3D modeling and graphics.

At first glance, offering a class on computer gaming seems rather frivolous. Many people view computer games as an idle form of entertainment. However, the entertainment industry is a multi-billion dollar industry with computer gaming forming a core component. In 2001, the computer gaming industry earned revenues in the range of nine billion dollars in the US (as opposed to eight billion dollars grossed at the box office for movies) [Eddy 2003]. Moreover, the distinction between computer gaming and the traditional movie industry is blurring with the reliance on 3D computer graphics in movies such as *Toy Story* and close tie-ins of new computer games to traditional movies. For example, the game *Enter the Matrix* incorporates a good deal of motion-captured video shot during the filming of the movie *The Matrix Reloaded*.

Putting aside the commercial importance of computer gaming, the educational importance of computer game design and creation lies in the fact that it provides an ideal framework for students to gain expertise and experience on a range of topics from computer graphics to software engineering. Given the popularity of computer games with college students, the class is extremely well received here at Rice and attracts the best students in computer science to the area of computer graphics. At Rice, this computer gaming class serves as an upper-level capstone course that performs several functions including:

- Exposing students to the practical realities of advanced topics in computer graphics and geometric design. For most games these topics include some type of polyhedral modeling, real-time rendering, collision detection, character animation and networking.
- Exposing students to the demands of building a real-time, networked application. Since computer games typically have high performance requirements, students are forced to focus on the performance of their code as well as correctness. Moreover, building a multi-player game requires students to confront the difficulties in writing, integrating and debugging real-time, asynchronous code.
- Providing students with experience in software engineering. Whereas previous classes in the CS curriculum focus on smaller individual coding projects, building a computer game involves a large team of students integrating various pieces of code developed during the current class as well as recycling code written in previous semesters.
- Generating a framework for integrating on-going research projects into the educational process. At Rice, student-produced games have served to demonstrate the practical applicability of research ideas. For example, the game *Beasts* (built in spring 2002) demonstrated the ability of Dual Contouring [Ju et al. 2002] to contour large grids in real-time (see figure 1).

Our goal in this paper is to provide a blueprint for others who wish to build a class that focuses on designing and building computer games. To this end, we discuss the algorithmic and programming aspects of building a computer game in the next section. In section three, we consider the problem of making the resulting computer game look "good". In particular, we provide an overview of the role of graphic design to building such a computer game. Next, we include a rough timeline for constructing and integrating the major components of the game. Finally, we conclude with a set of observations drawn from our experience of teaching this gaming class three times.

2 Computer game structure

In building a computer game, our approach is to decompose the game into a half-dozen main components and assign each of these pieces to a student (or pair of students). The student's responsibility is to implement these components during the first third of the semester and then integrate their components into the rest of the game. In this section, we give an overview of the decomposition used in our class. (Note that this decomposition while similar to that

*e-mail: {sschaefe,jwarren}@rice.edu

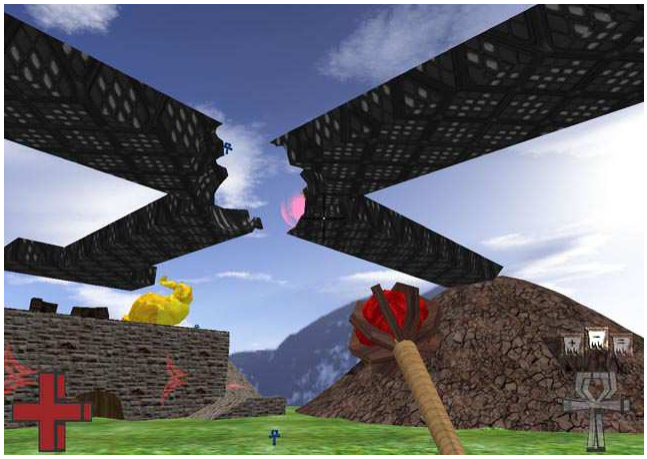

Figure 1: Destructive geometry in *Beasts* using Dual Contouring to model the geometry.

used in commercial game design and construction retains features distinct to the class's educational purpose.) Before we proceed, we first consider some relevant programming issues.

Long before work on the game starts, we must decide on what language the game will be written in. In the Rice CS department, the mainstream language taught in the lower level classes is Java. Since the real-time performance of the game is critical, even a factor of two slow-down due to language overhead is unacceptable. Therefore, we have chosen C++ (using *Microsoft Visual Studio* as the compiler) for its speed and ability to interface with publicly available libraries. This decision allows us to introduce some of the more beneficial aspects of the language such as templates, but also requires us to address other issues such as manual memory management.

To manage the code for a project as large as a computer game, we employ a source code control program. While a tool such as *PerForce* would be ideal, the class is limited to software that the University can purchase or acquire for free. As a result, we use *CVS* to manage the code repository for the class. This repository serves as a single distribution point for the code and helps manage concurrency issues that arise during development. Source code control also provides an archival mechanism so that code can be rolled back to a stable state if serious errors are introduced during the coding process.

The class also enforces a strict standard for adding code to the repository. Minimally, newly checked-in code must compile. "Breaking the build" is a serious offense and is discouraged through peer-pressure (the identity of the offender is announced to the rest of the class via email). Ideally, the code should also pass a series of runs testing its correctness before check-in. In practice, this method of testing is difficult to implement. Many factors influence the game and its state such as precise timings of input and even packet delivery over the network, both of which are problematic to control. Consequently, many of our tests are performed in a stand-alone version with networking disabled and are not usually sufficient to guarantee the program is in complete, working order. However, this strategy does catch many trivial errors that would normally cause great headaches for the rest of the development team.

2.1 Modeling

When distributing different components of the game to students, we try to give core components to students that are experienced and motivated to work on the game. Modeling is perhaps the most important piece of the game. The choice of modeling system dictates what type of geometry will appear in the game, the functionality of that geometry during game play and the way artists will manipulate and produce the geometry for the game. For the game to be successful, the modeling system must be completed and in place early in the development process.

Figure 2: Destructive geometry in *Hive Assault*. Left shows player standing in columns. Right, a hole has been created in the floor where the player has dropped through.

Previous versions of the class have used different representations for game geometry each with its own strengths and weaknesses. One representation is implicit modeling where game geometry is not explicitly stored, but instead computed as the contour $f(x,y,z) = 0$. Instead of explicitly storing f, we represent function f as a set of discrete samples on a uniform grid (or an octree). To produce the game geometry for rendering, we employ an algorithm such as marching cubes (MC) [Lorenson and Cline 1987] that processes each cell in the grid individually and extracts polygons to approximate the surface over those cells. One advantage that implicit modeling provides is that approximate CSG operations are fast and can be performed by using a Min/Max operator on the grid vertices. Real-time CSG allows the players to modify the game geometry interactively and offers a feature that is not found in most commercial games. Figure 2 shows two screenshots of a game called *Hive Assault* developed in the first incarnation of the class. On the left, a player stands in the middle of four columns. On the right, the geometry has been destroyed and the player has dropped into the chamber below allowing some of the "bugs" to crawl out of the tunnels underneath the floor.

One disadvantage to this approach is that a programmer must write conversion code to generate this implicit representation from a polygonal model. Also, the game cannot represent geometry that is smaller than a grid square, which limits the freedom of level designers. Furthermore, the level designer's job is also complicated by the fact that the geometry that appears in the game is dependent on how the imported polygons lie in the grid.

In the most recent game produced by this class, we elected to use a more traditional method of simply storing a "soup" of polygons. This approach yields the greatest flexibility in terms of modeling as arbitrarily complex geometry can be stored independent of any underlying grid. Moreover, the polygons and vertices can be explicitly annotated with information such as texture coordinates and normals. Popular modeling packages such as *Maya* are also able to export these polygonal models into standard formats, which can then be imported into the game. However, the downside is that we have sacrificed the constructive/destructive geometry that makes the implicit form so attractive.

Another advantage to conforming to popular practices is that we can adopt existing geometry formats from commercial games whose data formats are known. For instance, for the most recent game, *Time Bomber*, we decided to utilize the level format from *Quake III*, which immediately provided test levels that are publicly available. This choice alleviates a bottleneck in the coding process as the students responsible for modeling have more time to com-

plete the conversion process for the levels the designers generate. Furthermore, students that are assigned tasks such as rendering and collision detection are very dependent on having these levels for testing and can debug their code before the conversion process is ever completed. However, using existing level formats also brings the design decisions and features present in that data format, which can be unneeded or even unwanted.

2.2 Rendering

The students assigned to the rendering component have the responsibility of drawing the polygons generated by the modeling portion of the game. There are really only two choices for 3D API's (OpenGL and DirectX) and we haven chosen OpenGL both for its ease of use and portability. Since our primary bottleneck in game performance is rendering polygons, it is important to use vendor-provided extensions to increase the frame-rate. Unfortunately, different graphics card vendors provide different extensions. Luckily our target platform (the computers in the lab) is very well known and all the machines have identical hardware, which allows students to concentrate on optimizing performance for a specific graphics card. Students assigned to rendering are also responsible for the lighting in the game and writing any vertex or pixel shaders required by the look and feel of the game (for instance cartoon rendering). Any advanced effects such as shadows and per-pixel lighting fall under this category as well and students are encouraged to research and implement these methods if time permits.

2.3 Collision detection and physics

Typically the games that we create in the class are first-person games and, consequently, collision detection is very important for defining how players interact with the game. Our games require collision information for a variety of primitives. We approximate players in the game as axis-aligned ellipsoids since the player's geometry can be composed of thousands of polygons and is constantly changing. This simplification introduces errors in collision, but is rarely noticeable and makes the collision detection algorithm faster as the game must support many players interacting with the geometry in real-time. The collision system must also support ray intersection calculations with the geometry and ellipsoids. Rays are used for line-of-sight calculations and for weapons in the game. Particle effects can also use ray collision detection to interact with the geometry.

Collision detection must be very fast if the game is to run at playable frame rates. To alleviate concurrency issues, we perform all game play calculations on a server, which then updates the state of the client computers during the game. This server is responsible for all of the collision detection operations and must execute over a thousand collision tests per second using only a fraction of the total CPU. Maintaining these high frame rates requires a spatial partitioning data structure to enable fast lookup of the polygons that might collide with a specific primitive as testing all of the polygons is prohibitively expensive. In games where we use an implicit representation to perform real-time CSG, the polygons in local areas of the geometry can change at any time. Therefore, we cannot use a data structure that requires any significant processing time to construct such as a BSP tree. Luckily, the data in these representations resides in a grid or octree and naturally provides a spatial partitioning structure. The programmers must then perform ray or ellipsoid collision detection against some subset of the polygons.

In our most recent game, *Time Bomber*, we store a "soup" of polygons for the game geometry. To effectively process this geometry, we decided to use a more traditional BSP tree [Fuchs et al. 1980] to partition the data. BSP trees partition space into convex regions using a binary tree. Branch nodes contain a splitting plane (usually chosen from the set of polygons) that separates space into two pieces and the geometry is distributed to each child corresponding to the front and back sides of the plane. Leaf nodes represent the convex regions of space and contain information denoting whether the region is solid or empty.

To build the BSP tree, we require that the geometry is manifold, oriented and not self-intersecting so that the polygons locally describe whether the surrounding space is inside or outside of the solid. Unfortunately, these constraints limit the types of levels the designs can generate within the same polygon budget. For example, our level designers were content to allow columns to penetrate into the floor, but these columns must be unioned with the rest of the model to generate a non-intersecting manifold and drives the polygon count of the level up despite no visual change. In fact, we found the only way to reliably ensure that our geometry satisfied these constraints was to limit the operations the level designers could perform to functions such as CSG, extrude, etc. Since the polygon count increases dramatically with these operations for the same visual quality, we opted to use two sets of polygons: one for constructing the BSP tree, the other for display. Unfortunately, this decision adds complexity to the design process. Furthermore, BSP construction is fraught with numerical issues arising from nearly coplanar faces and sliver polygons.

Point and ray collision are fast and easy to perform using BSP trees. To perform ellipsoid collision detection we use an approximate algorithm [Melax 2000] that reduces the problem to point classification by moving each of the faces bounding the solid convex regions outward. Altering the planes in this fashion does not generate a true offset surface and the error increases dramatically as the dihedral angle becomes large. These errors lead to invisible walls in the collision detection where players seemingly cannot move through open space. To combat these problems, we are forced to add beveling planes on edges of convex regions that are orthogonal to the direction of maximal error. The beveling planes reduce the amount of error caused by offsetting the convex regions, but do not eliminate the error. We found this solution difficult to implement and not extremely robust. At the end of the class, BSP collision was an order of magnitude faster than ellipsoid collision with simply polygonal information. Despite the problems with BSP trees, this experience was educational for the students as algorithms that seem simple in theory can be burdened with unseen difficulties in practice.

The student(s) assigned to the physics component of the game typically must work closely with the group working on collision detection. These students do not simply implement Newtonian physics, but must also define how entities react to one another in the game. For instance, in *Time Bomber* the players throw bombs that bounce off the geometry. Physics is responsible for making these bombs bounce, controlling how much they bounce and providing the necessary change in trajectory when two bombs collide.

2.4 Animation

In our games, animation controls how the actions of other players are visually conveyed. For example, when a player fires a gun the animation system alters the visual appearance of that player to display the specified motion. While this feature may not be critical to game play, the game is substantially less playable without the visual cues animation provides and, hence, we view it as a requirement. Students have implemented two different options for animating characters over the past three years: skeletal animation and key frame animation. Each approach has different strengths and limitations discussed shortly. The game then transitions between different animations using a state table that is indexed by events in the game and the current animation being performed.

Skeletal animation [Magnenat-Thalmann et al. 1988] is an ani-

mation technique that uses a graph structure to represent the "skeleton" of the character with each edge (called a "bone") in the "skeleton" representing a local coordinate frame. The character is animated by representing vertices of the model in terms of the local coordinate frame of a given "bone". Animators then deform the skeleton structure, which induces a deformation on the vertices of the model and causes the character to animate. In areas where the model bends substantially such as the elbow, vertices can be represented in multiple coordinate frames and affinely blended together to generate more aesthetically pleasing animations. This system creates small animation files since only the graph structure needs to be stored for the animations and provides the animator with a simplified interface for constructing these animations. Moreover, this method lends itself well to dynamic animations in game such as having the characters recoil in different ways from being shot at various locations without using pre-scripted animations.

Constructing the skeleton for a model and assigning the vertices to the bones can be a non-trivial task. Our solution has been to use publicly or commercially available tools to construct the animations. One such tool that we have used is *Poser*, which contains a sizable database of pre-existing animations. However, these animation tools do not export animations to any common format that contains all of the information necessary for skeletal animation. For example, these tools usually contain heuristics for automatically assigning weights to the vertices given influence bounds of bones but do not expose the assigned weights to the programmer. This lack of information requires the programmer to infer the missing data and, consequently, our animations do not always appear exactly how the animators intended causing further frustration.

In contrast, key frame animation does not use a skeleton for animation. Rather the only information provided is the position of the vertices of the character at specific points in time. To perform the animation, the program simply blends the position of the vertices together as a function of time between two different frames. The advantage of this approach is that the data is simple and any animation program can export models at specific instances in time. Furthermore, this system frees animators from any specific skeletal constraints and can allow for greater flexibility in the animations created. However, key frame animation does not lend itself well to performing dynamic animation and generates substantially larger animation files since the size of the file is proportional to the complexity of the model and not of the underlying skeleton.

2.5 Networking

Students assigned to networking have a difficult task as their code must be written and debugged early in the semester. Our games begin as stand-alone games, but are really designed to be played with many players networked together. Without a well-defined networking interface for communication, progress in the game can be limited during the first part of the semester.

To make matters more complicated, we run the networking system on a different thread than the game to assure low latency in packet delivery. This decision requires careful thought as to where mutexes need to be placed in the code to avoid concurrency issues. To this end, students have developed a networking infrastructure that restricts the way the separate threads communicate to a single shared queue as well as minimizing the amount of code required in the portion of the code locked by the mutex. In this design, the networking thread constantly delivers messages into a holding queue. When the game thread is ready to accept messages, it locks the queue, switches the active queue with an empty queue and then releases the lock. Now the game thread has exclusive access to the message queue while the networking thread can continue to deliver messages to the switched, empty queue. The students have found this design alleviates the concurrency problems associated with networking and is simple to implement.

The games that we create have typically used a client/server architecture for communication where the clients are passive entities directing input to the server and displaying the current state of the game. The server itself actually performs all of the game computations and acts as the communications hub for the players. We use two different networking protocols for communication in the game: TCP/IP and UDP. The game uses TCP/IP for guaranteed messages that are essential to the game such as player death. The problem with this protocol is that it has a high latency and cannot be used for messages such as player positions that need to be sent up to sixty times per second. In contrast, UDP does not guarantee delivery or order of packets, but does provide very low latency. Therefore, we use UDP in the game for messages that need to get to their destination quickly but do not affect the overall game if some are dropped (for instance particle effects).

2.6 Other components

Besides the programming components that we have already mentioned, there are several other tasks that are not necessarily critical to the game but improve the game experience. For instance, many of our menus and configuration options tend to be command line based because they are simple to program. However, from a user's perspective, this interface is less than ideal. In our last game we assigned two students the job of creating a menuing system for the game. This task required that they work closely with a 2D artist that could provide them the required art for the menus as well as transparency masks and fonts that fit the look-and-feel of the game. The result was a much more professional looking game though the functionality did not change significantly.

Another component that is typically found in commercial games is artificial intelligence (AI). All of our games have been from the first-person perspective and we have attempted to construct other entities in the game to serve as the antagonist or simply to enhance the game play experience by adding other creatures that the players can interact with. Unfortunately, AI requires deciding not only how computer controlled entities react and their strategies, but determining how they move in the game as well. For our first two games, we provided players with the ability to perform constructive and destructive geometry. Path planning in these types of environments is extremely difficult and students had many problems implementing the movement of computer controlled entities. Our last game contained static geometry and we had more success with path planning, but our games typically degenerated into large, multi-player games where the AI played a fairly insignificant role. Given the difficulty of implementation and the small benefits for a multi-player game, AI is considered an add-on to the game, but is usually dropped from the final implementation.

Sound is also a task that we assign to students that does not directly affect the game's playability but does enhance the game play experience. For instance, the sound of footsteps or gunfire can give the player clues to where other players are as well as providing a deeper level of immersion in the world of the game. Unfortunately, the difficulty with sound is not programming, but obtaining the sounds needed for every situation. The sound effects that our games do have are either found from public sites on the internet or recorded using very rudimentary methods. For programming, we typically use an external library such as *FMOD*, which offers decoding and playback of many popular sound formats.

3 Graphics design for games

To build a visually appealing game, good graphic design is critical. For most commercial games, the manpower devoted to art greatly exceeds the total programming effort. Given that the class is

typically composed of computer science students, not art students; achieving good graphic design is usually more difficult than programming the game itself. In each of the games developed in the first three versions of the class, we devoted an increasing amount of resources to graphic design with only a modest increase in the visual quality of the game. In our experience, the key to good graphic design is locating students with artistic talent and then providing them with access to and training in state-of-the-art 3D graphics design tools. For example, in the latest version of the game, we recruited Architecture students from outside of the Computer Science department to add more graphic design expertise into the class. Together with the several CS students, we then trained these students in the basics of 3D modeling and had them focus solely on 3D graphic design during the class.

3.1 Level design

Designing levels for a game such as a first-person shooter requires not only geometry, but also realistic textures for the geometry. In practice, building a custom tool to facilitate this process is not possible in the course of a semester since the tool itself would be more complicated than the game. Instead, we have adopted the strategy of using a commercial modeling tool to design game geometry and then export this geometry to the game engine. In the most recent game, we have focused on using *Maya*, a state-of-the-art 3D modeling environment. *Maya* has many advantages; it has a natural interface that is easy to learn, extensive functionality and a scripting language *MayaScript* that facilitates exporting geometry designed in *Maya*. Note that from an educational point of view, exposure to a tool like *Maya* gives students more experience in the everyday practice of computer graphics.

In our experience, designing a game level breaks down into three subtasks:

- Constructing a polygonal representation for the level.
- Assigning texture coordinates to the vertices of the polygons.
- Generating textures for the polygons.

The first task, building a polygonal representation for the level, is one of the core functionalities of *Maya*. We designed components of the levels using mainly polygonal mesh operations and combined these components using CSG operations. However, we recommend that one avoid taking CSG combinations of separate meshes that share coplanar polygons since this operation is unstable and can often lead to meshes with non-manifold geometry. (Non-manifold meshes cause serious difficulties during BSP tree construction.)

Maya also provides excellent support for assigning texture coordinates. Specifically, it allows the user to specify a subset of the polygons in the mesh and then to project these polygons onto a user-specified plane. This planar mesh can then be exported to a program such *Adobe Photoshop* and textured using *Photoshop*'s 2D image processing capabilities. Of course, generating the actual textures used for the level is a full-fledged problem in itself. In practice, the students scoured the web for free textures that provided the desired effect.

3.2 Character design

When generating a character for the game, there are two possibilities for procuring models. The first (and easiest) is to simply find an existing model available on the web and to make modifications to that model. This solution is usually motivated by lack of artistic manpower and also has intellectual property issues. The other solution is to have students build models themselves, which can require

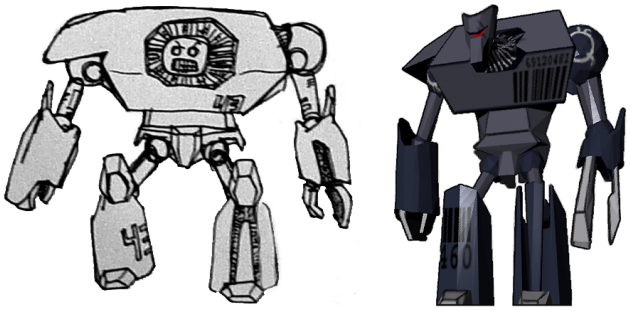

Figure 3: Character sketch created for *Time Bomber* (left). Actual model generated for the game (right).

substantial effort as modeling humanoid figures is much more difficult than constructing levels composed of rectilinear solids. However, we have found that the later solution is possible in the course of a semester. The design of these characters can also be simplified by moving away from humanoid objects to more rigid shapes such as robots or spaceships (see figure 3).

The artists that create characters are not as constrained in their development as the level designers in that there is more topological freedom. However, these students must observe a strict polygon budget as the game must be able to handle many players animating all at the same time. Therefore, we limit these models to approximately 1500 polygons. The students typically create models without respect to this limitation and use software such as QSlim [Garland and Heckbert 1997] to reduce the model to the specified number of polygons. This strategy imposes an artificial limitation on the connectivity of the models because the simplification software has difficulty simplifying visually closed surfaces composed of abutting open patches. Therefore, modeling primitives such as NURBS that generate "T"-vertices cannot generally be used if simplification must be performed later.

Once the geometry for a character is complete, the students must create textures and assign texture coordinates to the surface. Textures are used to add detail such as facial features that cannot be incorporated into the geometric model due to the polygon budget for each model, and texture coordinates are assigned in a similar manner to level design. Finally, the complete model is animated in *Maya* by annotating the geometry with skeletal information and animations are extracted using *MayaScript*.

3.3 Item design

Items such as weapons and powerups also play an important role in our games. To obtain models for these objects we try to find freely available models on the internet that suit the purposes for the game or, lacking an alternative, purchase low cost models with the funds available for the class. Another method is to have the students construct these objects themselves. However, our games usually require a fair number of different types of weapons and powerups and it is not always feasible to model each item during the course of a semester. One method that we have had success with is to create two-dimensional icons for the powerups in the game and then to extrude the boundaries of these images into a three-dimensional object. This method automatically generates the texture coordinates for the objects and creates visually appealing models.

3.4 Other graphical elements

Besides three-dimensional modeling, students can be assigned to more traditional two-dimensional art if there are any talented artists

in the class. These students can work with the programmers assigned to the user-interface to produce menus for the game (which includes alpha masks for transparency) and fonts to the programmers' specifications. Students talented in drawing have also designed images for our web page as well as title screens for the games.

If the class contains any musically gifted students, these students can be encouraged to generate music for the game as well. In our most recent game, two of the students were able to create several original soundtracks for background music in different levels. This addition enhances the game play experience and imparts a more professional quality on the final product.

4 Class structure and timeline

Having covered the basic elements of programming and graphic design needed to construct a game, we next consider the organization of the gaming class and a rough timeline for completing the various portions of the game. In a typical fall, between 30 and 50 CS students take the introductory computer graphics class. Around Thanksgiving, we give a demo of the previous spring's game to the computer graphics class and encourage the top students in the class to apply to be included in the next semester's gaming class. Restricting the enrollment in the game class has several beneficial features. First, restricting enrollment allows the instructors to cull students whose performance in the introductory class is substandard. This process serves to eliminate those students who think computer gaming is "cool", but who are not ready for the high workload required by the class. Second, the remaining students view being part of the class as an honor and tend to work harder in the class as a result. (One student compared the class to working for a startup company, but without the pay.) One consequence of this view is that peer pressure, instead of traditional grades, serves as the primary motivating force in the class. Students are strongly encouraged (and supported) by their classmates to complete their individual components in a timely manner since building a successful game requires all pieces of the project to be completed. In the rare case when it becomes clear that a student is unable to complete his or her component, the instructors encourage the student to drop the class and the remaining members of the class finish the component.

In a typical instance of the class, we have between 10 to 15 students with 6-8 students focusing on programming and 4-6 students focusing on graphics design. The most recent class also included several Architecture students whose primary interest was graphic design. Various programming and graphics design components are assigned to the student based on their interests, talents and willingness to undertake the project. One critical part of this assignment process is that proactive students (which tend to perform better independently) be assigned the core infrastructure of the game such as the level modeling and networking. Reactive students that require more direction can then be assigned more peripheral tasks such as building a user interface.

In our structuring of the class, the authors' role is less that of instructors and instead more that of managers. Warren's role as professor was to serve as a senior manager that lays out the high-level conceptual design of the game and a timeline for developing/integrating these components of this design. This design would include choices such as using a particular modeling technique to represent game geometry (say implicit modeling). The students themselves would then be responsible for learning the appropriate material needed to implement these concepts as well as developing interfaces for connecting their code to the game. During the course of the semester, Warren serves the role of overseer to ensure that the students are adhering to the timeline and also serves as a problem solver assisting the students in solving various technical problems that arise during the course of their projects. Schaefer, as lab assistant, performs the role of coding czar to set and enforce coding standards, tutor students in advanced programming skills such as OpenGL and assist student in tracking down bugs that arise from the interaction between several components.

During the spring semester, class meets for 2 hours on MWF afternoons and typically consists of a short tutorial on some topic of interest (e.g. modeling in *Maya*), followed by a progress report of various components of the game and finally time for individual work that may include programming, integration, debugging or play testing. Students are encouraged to work for all or most of the two hour class period (even if not directly working on their project so that other students may interact with them if necessary). At Rice, class is held in the Symonds II laboratory (http://cohesion.rice.edu/naturalsciences/citi/symondslabs/index.cfm), a custom-designed space that serves as a hybrid classroom/programming lab. The lab contains 20 PCs equipped with high performance graphics cards and provides a very comfortable environment for learning and working. Since students also work extensively outside of class time, the lab serves as a common meeting place for students to collaborate on class activities (meetings, debugging, play testing).

While each game has its own timeline in terms of implementing and integrating the various components that makeup the game, there remains a common high-level structure to these timelines. Perhaps the most important point to note is that a 15 week semester is an extremely short time span to build a reasonably detailed game. As a result, starting early on the planning and design of the game is critical. Typically, we begin holding weekly meetings to discuss game design late in the fall semester prior to the start of the spring semester. These discussions allow the class to have input on the type of game being developed (a first person shooter, a real-time strategy game, etc.) and start the process of assigning the various game elements to individual students. One approach that we took in designing the game was to build a variant of an existing game, but include several unique hooks to distinguish the game from current games. For example, the first game *Hive Assault* was essentially a standard first person shooter with two unique hooks: destructible geometry and real-time fluid dynamics.

At the end of fall class, the instructors make a preliminary assignment of students to game components. Then, over Christmas break, students are responsible for completing the groundwork necessary to jumpstart their project in the spring. For example, a student might read a paper on implicit modeling or review an existing portion of the code base that will be reused. Having student actively work on their projects during the first few weeks of class is key since most students have a relatively light workload from traditional classes during this time.

During the semester, the class timeline is broken into three parts, each corresponding to a five week period.

- *First five weeks: Implement various components*. Typically, we require a rough prototype of the geometry engine and networking to be finished early in this period. The graphic design group should have exported a simple test level (untextured) to the geometry engine. Having these components available gives the class its first taste of what the game will look like and allows other components to be tested more thoroughly using this core infrastructure.

- *Second five weeks: Integrate various components*. In particular, students add collision detection/physics, simple character animation and a preliminary user interface to the game. The graphics design group should have at least one high-quality, textured game level. The group should have also produced at least one test character for the animation system.

- *Third five weeks: Debug and play test game*. Students typically enhance the character animation and add AI to the game. Various graphical design elements are added to the game and its user interface. The graphical design group prepares several levels and several full-fledged characters for the animation system. Game play is adjusted to enhance the game's "fun".

While this timeline is not too precise, we typically set concrete, weekly goals for the various components in the game. For example, in character animation, a sequence of short-term goals might be: display a character as a cube, move cube via simple keyboard commands, replace cube by a skeleton, add infrastructure for skeletal animation, build animation for more complex actions such as jumping, add polygons to skeleton, add texturing to polygons. These weekly deadlines are important in partially mitigating the inevitable delays that result from students underestimating the difficulty in their project or being swamped by work from other classes.

5 Conclusions

Having taught three cycles of this gaming class, a number of lessons about making this class a success have become apparent.

- Perhaps, the most import lesson is that following good software design and coding practices is essential to the success of the game. All three of the finished games included more than 75K lines of C++ code. While a project of this size is small by industrial standards, the code base for the game is the largest that most students will have ever worked on. Maintaining good coding standards allows other students to reuse, understand and modify code written by other students. Perhaps the most important educational feature of working on such a large project is the basic fact that writing bad code will almost certainly come back to haunt the student who wrote it. Once the student's peers have had to integrate, modify or debug this bad code, peer pressure serves as powerful tool to improve one's coding practices.

- Second, the size of the project presents the students with the novel (for them) situation of having to debug a piece of software which no one person may totally understand. In practice, the class spent a large part of the last five weeks tracking down extremely subtle bugs in the game. These bugs were especially difficult to identify and remedy due to the networked nature of the game.

- Next, "eye candy" is a very important facet of a computer game. In the first game, our class focus was almost entirely on programming. The resulting game, while incorporating several interesting features, was not particularly pretty. After demoing this first game to a commercial game designer, the designer pointed out that real commercial games typically have two graphic designers for every programmer. Given our class is a computer science class, we have attempted to address this problem in two ways. First, we have actively recruited Art and Architecture students for the class. Second, we have acknowledged the limitations of our graphics design skills and instead explored ways of using humor to compensate for bad graphics design. For example, instead of trying to generate realistic sound effects for a game such as the sound of a cow, we instead substituted a student saying "moo" into a microphone. The result effect causes the user to laugh instead of considering the primitiveness of our sound effects.

- The lab environment for the class plays a crucial role in the success of the class. Staging the class in a physically comfortable environment is helpful in encouraging a "lab culture" to form. This culture encourages students to assist each other in the learning, design and programming necessary to build the game. Having a comfortable work area also encourages students to congregate outside of class to work on the game.

- Finally, grades in the class should be a non-issue. Students should enroll in the class not to earn an A, but instead to be part of a fun and enjoyable learning experience. In our experience, peer pressure from other students in the class is usually sufficient to motivate a student to complete their part of the game.

In future versions of the class, we would like continue to increase the artistic component of the game. Our experiences in previous classes have helped us determine what portions of the art are problematic and we can plan accordingly. We also believe that we can increase the complexity of the games by leveraging code designed in previous versions of the class to avoid having to reconstruct core components (such as rendering and networking). This reuse should give the programmers more time to focus more on game play rather than getting the game to a baseline, working state.

References

EDDY, A. 2003. On with the show... again. *http://www.gamespy.com/bizbuzz/january03/bizbuzz49/* (January).

FOLEY, J., VAN DAM, A., FEINER, S., AND HUGHES, J. 1990. *Computer Graphics: Principles and Practice, 2nd Edition*. Addison Wesley.

FUCHS, H., KEDEM, Z., AND NAYLOR, B. 1980. On visible surface generation by a priori tree structures. In *Proceedings of SIGGRAPH*, ACM SIGGRAPH / Addison Wesley, Computer Graphics Proceedings, Annual Conference Series, 124–133.

GARLAND, M., AND HECKBERT, P. S. 1997. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH*, ACM SIGGRAPH / Addison Wesley, Los Angeles, California, Computer Graphics Proceedings, Annual Conference Series, 209–216.

JU, T., LOSASSO, F., SCHAEFER, S., AND WARREN, J. 2002. Dual contouring of hermite data. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, 339–346.

LORENSON, W., AND CLINE, H. 1987. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics 21*, 4, 163–169.

MAGNENAT-THALMANN, N., LAPERRIERE, R., AND THALMANN, D. 1988. Joint-dependent local deformations for hand animation and object grasping. In *Graphics Interface*, 26–33.

MELAX, S. 2000. Dynamic plane shifting bsp traversal. In *Graphics Interface*, 213–220.