Learning through Game Modding

Conference Paper *in* Computers in Entertainment · January 2005

DOI: 10.1145/1111293.1111301

CITATIONS
109

READS
543

2 authors:

Magy Seif El-Nasr
Northeastern University
163 PUBLICATIONS **1,892** CITATIONS
SEE PROFILE

Brian K Smith
Drexel University
64 PUBLICATIONS **1,059** CITATIONS
SEE PROFILE

Learning Through Game *Modding*

MAGY SEIF EL-NASR AND BRIAN K SMITH

The Pennsylvania State University

There has been a recent increase in the number of game environments or engines that allow users to customize their gaming experiences by building and expanding game behavior. This article describes the use of modifying, or *modding*, existing games as a means to learn computer science, mathematics, physics, and aesthetic principles. We describe two exploratory case studies of game modding in classroom settings to illustrate skills learned by students as a result of modding existing games. We also discuss the benefits of learning computer sciences skills (e.g., 3D graphics/mathematics, event-based programming, software engineering, etc.) through large design projects and how game design motivates students to acquire and apply these skills. We describe our use of multiple game modding environments in our classes. In addition, we describe how different engines can be used to focus students on the acquisition of particular skills and concepts.

Categories and Subject Descriptors: K3 [**Computing Milieux**]: Computers and Education; I.6 [**Computing Methodologies**]: Simulation and Modeling; D.m. [**Software**]:Miscellaneous

General Terms: Experimentation, Design

Additional Key Words and Phrases: Games and education, game engines and classrooms, learning and design

INTRODUCTION

In the late 1990s, game developers began to separate gaming experiences (e.g., rules, behaviors, characters) from the underlying engines that power them. These game engines allow designers to create new game behaviors and graphics by plugging into reusable architectures that handle polygon rendering, camera control, lighting, and so on. Many popular game engines come with scripting languages that allow users to modify their behaviors, create new worlds for exploration, or even modify existing games into completely new ones. This process, often referred to as *modding*, still requires designers to understand how to communicate with game engines, but the overhead of creating a working product is significantly less than working without the functionality that these engines provide.

Modding offers a number of advantages over designing games from scratch [Emmerson 2004]. The time and costs related to video game development are enormous, preventing most individuals from being able to create games that resemble those built by corporate designers. However, modifying existing games is within the reach of many game players, as companies provide tools to allow designers to tweak their games. Since modding begins with popular, proven game concepts, the resulting variations are more likely to resemble games that players/modders are accustomed to than if they had to build entire game infrastructures on their own.

Authors' address: Magy Seif El-Nasr, School of Information Science and Technology, The PennsylvaniaState University; Brian K Smith, School of Information Science and Technology & College of Education, The Pennsylvania State University; email: [magy, bsmith]@ist.psu.edu

In this article, we discuss the use of game modding as a pedagogical activity. Although much of the complexity of creating games can be eliminated by modifying existing games, there are still opportunities for designers to learn about concepts such as 3D graphics, vector geometry, event-driven and object-oriented programming, artificial intelligence, and computational and aesthetics principles that are tied to game development. In fact, modders may be able to focus on learning these fundamental design skills *because* game engines and their tools eliminate much of the overhead associated with building convincing products. We begin by discussing the benefits of learning by designing computational artifacts. We then describe the computer science skills and concepts required to mod games through two case studies of students working with game engines in classroom settings. Specifically, we report observations of high school and college students enrolled in game design courses at The Pennsylvania State University and present examples of their games and the skills they acquired to create them. Through these examples, we will suggest how different game modding environments can be used to help students develop particular skills and concepts.

LEARNING BY DESIGN

Modifying existing games to create new ones is a design activity that we believe may have educational benefits. There are several reasons why design tasks are useful for learning content, skills, and strategies [Puntambekar and Kolodner 2005]. Design activities provide meaningful, engaging contexts for students to explore skills and concepts and understand how they can be applied in the real world. During the design process, skills such as analysis, synthesis, evaluation, and revision must be used, providing opportunities for learning content and metacognitive skills such as planning and monitoring. Students can receive ongoing feedback from peers and experts when constructing working artifacts. Feedback also comes during the process of construction as students work to understand how and why their designs fail, can be optimized, and so on. Finally, real design problems have multiple solutions, allowing students to see and evaluate alternatives. This leads to iterative activities where students incrementally build, evaluate, discuss, and revise their constructions.

Design, and learning from the process, can occur in many domains with different types of construction activities, but we will focus on the design and development of game software. Since the development of the Logo language in the 1960s, educational researchers have investigated ways that programming computers can facilitate learning about mathematics, computation, and more general planning skills [Lehrer 1986; Papert 1980; Pea et al. 1987; Resnick and Ocko 1993)]. Seymour Papert, the father of Logo and perhaps the strongest advocate of programming as a pedagogical pursuit, used programming as one example of a larger theory of learning called *constructionism*.

Constructionism involves two activities [Papert 1980]. The first is the mental construction of knowledge that occurs with world experiences, a view borrowed from Jean Piaget's constructivist theories of learning and development. The second is a more controversial belief that new knowledge can be constructed with particular effectiveness when people engage in constructing products that are personally meaningful. Those products could be dollhouses, Logo programs, or particle accelerators. The important issue is that the design and implementation of products are meaningful to those creating them and that learning becomes active and self-directed through the construction of artifacts.

Constructionist approaches to using video games in education involve students designing and developing their own games. For instance, Harel's work in elementary

schools demonstrated children working for prolonged periods on the creation of educational games using the Logo programming language [Harel 1991]. Kafai [1994] noted similar engagement as students developed their own games, and she also tracked their abilities to incrementally create, evaluate, and revise their designs over time. Hooper's longitudinal study of software development in schools showed students expressing notions of cultural identity in their programs—ideas that were not likely to find expression if students had just played existing games [Hooper 1998].

These studies used Logo as the primary programming language, but a number of programming environments have been created to help novices learn by designing and implementing working computer programs [Conway et al. 2000; Ingalls et al. 1997; Repenning and Ambach 1997; Resnick 1994; Smith et al. 1994; 2000]. Toolkits like these have been used to help novices turn design ideas into working prototypes as well as to learn computational fundamentals.

Numerous universities have adopted game design into their curricula in hopes of increasing undergraduate enrollment in computer science programs [Angiolillo 2005]. The belief is that students' familiarity with games can be used to motivate computer science learning and attract and retain future generations of computer scientists. Students enrolled in experimental computer science classes that used game design averaged higher grades than control classrooms. Additionally, 88% of the game students continued in the major compared to 47% for the control groups [Moskal et al. 2004]. Combining constructionist pedagogy with game design seems to help some students dig deeper into computer science fundamentals.

The case studies described in this article can be considered another example of using games to teach computer science skills. Time, cost, and expertise are significant barriers to experimenting with video game design in educational settings, but customizing existing games may reduce the difficulty and make it possible for learners to create credible prototypes. Therefore, we have focused on understanding if and how game modding can lead to the same types of personal learning experiences described by Papert and others, and what types of skills can be acquired using different game engines.

MODDING IN CLASSROOMS

Since the release of Doom in the late 90s, game enthusiasts have been eager to customize their maps and modify their gaming experience by modifying the game through programming or scripting. Lately, the modding community has increased tremendously, with more gamers eagerly building and sharing their customized game experiences [Hyman 2004]. For example, Will Wright reported receiving over 74,000 character and object models built by people for the Sims [Wright 2004]. Another example is Counter Strike, a game mod built using the Half Life game engine. This mod was so popular that it is now included with the Half Life 2 game release.

Providing engines or editors with games is becoming more popular. Many games have released engines or tools that allow users to modify game maps, mechanics, events, and behavior. Examples include War Craft III, Unreal Tournament 2003 and 2004, Half Life, Half Life 2, and Morewind. Game modding has also become a good source of game industry recruiting and game prototyping.

In this section we describe two classroom sessions where game modding played a major role in helping learners work with computer science concepts. Our evaluations of these sessions are formative, relying on our observations of students performing game design for extended periods. Although these observations are exploratory, we believe that

the descriptive results we present may be helpful to others looking to implement game design in introductory and advanced computing curricula.

Game engines differ in their flexibility and the skills needed to use them. Discussing the differences between all engines in the market today is beyond the scope of this article. Instead, we describe three different engines—WarCraft III, Web Driver, and Unreal 2.5 Engine—and how they were used in our case studies.

FIRST SETTING – GAME DESIGN WORKSHOP

Classroom Setting

Our first case study describes a class on game design and programming for high school students as part of the Pennsylvania Governors School for Information Technology (PGSIT). Twenty high school students interested in IT enrolled in this summer class. The course was held on three consecutive days for three hours a day. The main aim of the course was to engage students in teams of two to build a game in three days.

Process

Since this was a short class, we needed an easy-to-learn, rapid prototyping engine. We chose the WarCraft III engine because it provides simple visual methods for customizing game templates and requires little or no programming skills to create interesting designs. The game engine is composed of several editors: the world or map editor, trigger editor, character editor, etc. The editors are implemented as visual programming tools that allow designers or users to visually customize game behavior, including character behavior, game map, and game play. The map editor (shown in Figure 1) is specifically used to create or customize maps for the game. As illustrated in the figure, the editor is composed of several visual tools that allow users to place different types of terrains, objects, and characters within the scene. Users can also create or identify special regions for triggering specific game behaviors. The trigger editor (shown in Figure 2) allows users to modify game behavior when a specific event or condition occurs. For example, one can include a trigger to start a specific song once the player enters a specific region in the map. A typical trigger consists of an event or a condition and an action. The event and condition part of the trigger specify when the trigger will become active. When the trigger is activated, the actions specified in the action part of the trigger will be executed. Therefore, creating several triggers can develop a different game. The game engine

Fig. 1. Warcraft III's world editor.

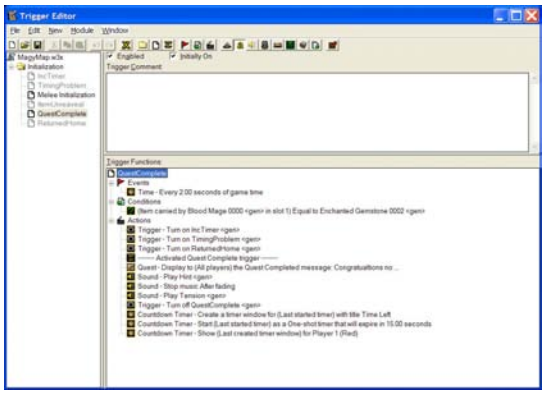

Fig. 2. WarCraft III's trigger editor.

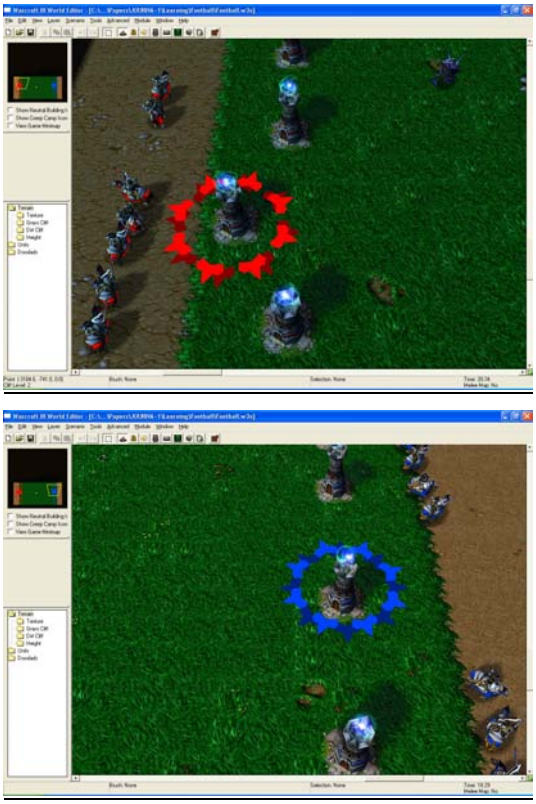

Fig. 3. Students created these graphical levels for their football game.

typically checks all user-specified triggers in parallel. The engine checks if the trigger condition and event are met: if they are, the game engine will then run the trigger actions.The students were asked to each find a partner and work together to develop a game concept on the first day. On the second and third days, they developed prototypes

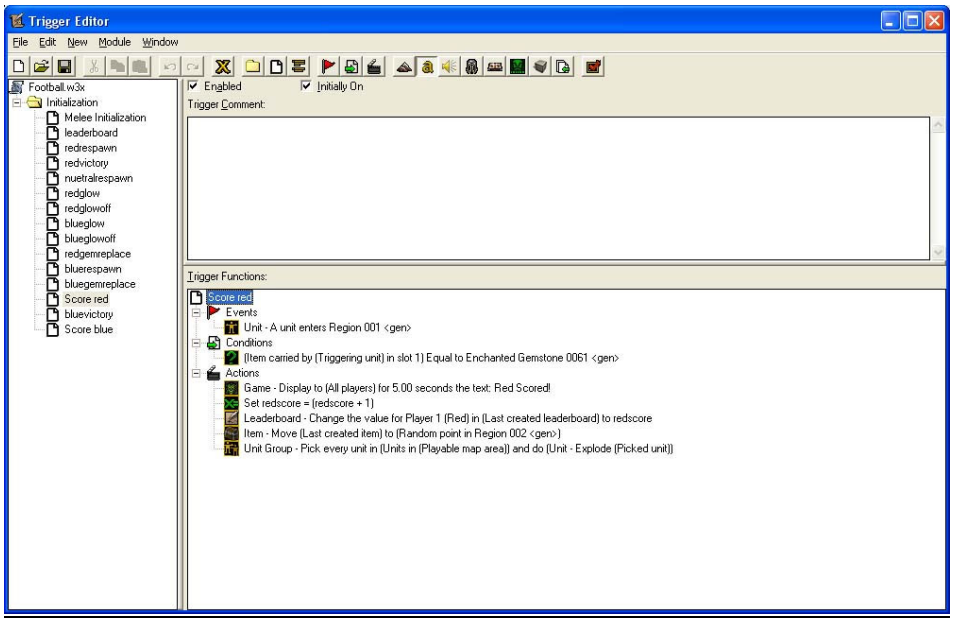

Fig. 4. The scoring trigger created by the students for the football game.

of their game using the WarCraft III engine. The games and ideas were critiqued during team meetings with the instructor and a graduate student with experience in game design. These critiques occurred informally as the instructors observed student progress.

The best way to illustrate the concepts that students learned is to look at the games they created. For instance, one group created a football game. Transforming the original WarCraft III game from a military strategy game to a sports scenario involved extensive modifications to the engine's rules. The students borrowed the team rules from the original game but developed their own graphical map and football characters (Figure 3). They repurposed an existing item, a treasure chest icon, to be the ball.

The students had to create new rules to implement scoring actions while removing those that allowed characters to use weapons. Figure 4 shows an example rule that they created to implement a touchdown in WarCraft III's visual trigger editor. WarCraft III is an ideal game engine for novice programmers. It also provides a good and quick engine to work with, considering the short duration of the class. Although modding takes place with highly structured visual tools, students have to develop some notions of game design and computation in order to create working products. In this example, students were able to create a simple graphical map and characters to create the setting for a football game. They also had to develop some understanding of production rules and Boolean logic in order to create the game's scoring rules. Building the scoring system also required them to understand how WarCraft III's rule system works. For instance, they had to be explicit about marking map regions where events could occur, recognize if the ball is in one of these locations, and specify which player is carrying the ball. Accomplishing these three tasks requires conceptual knowledge of unique identifiers (variables) and their use in

identifying regions and positions on the map. They also had to understand conditional tests when comparing the position of the ball to these specific map regions.

The football game went through three different iterations with instructors providing feedback to enhance the students' work. This allowed the students to develop their understandings of user interface and game design issues. For instance, in the first iteration, the students created a map that was visually complex, but it detracted from the goal of the game. They corrected this in their second version after receiving tips on how to strike a balance between aesthetics and game play.

Another student team modified WarCraft III to create a Tetris-like game. This introduced more complexity, as the students had to develop mathematical functions to rotate and translate puzzle objects. Tetris also involves more modifications to the original WarCraft III game than the earlier football example. The level design was simple, a large room to mimic the Tetris playing field (Figure 5). Several triggers and functions were added to rotate and translate puzzle units depending on user inputs. Figure 6 shows one of their functions that captures keyboard events in order to move the puzzle pieces. The figure shows several triggers on the left. We have expanded the *rotate* trigger in order to show an example trigger. This trigger in particular registers another trigger that sets rotations on keyboard events. Students were able to successfully create these triggers through consulting instructors and web resources, including message boards and tutorials on setting functions and triggers within the WarCraft trigger editor.

OBSERVATIONS

Game design, like software design, often involves many iterations to get the game concept and mechanics right. In the class, we first introduced different theories of game

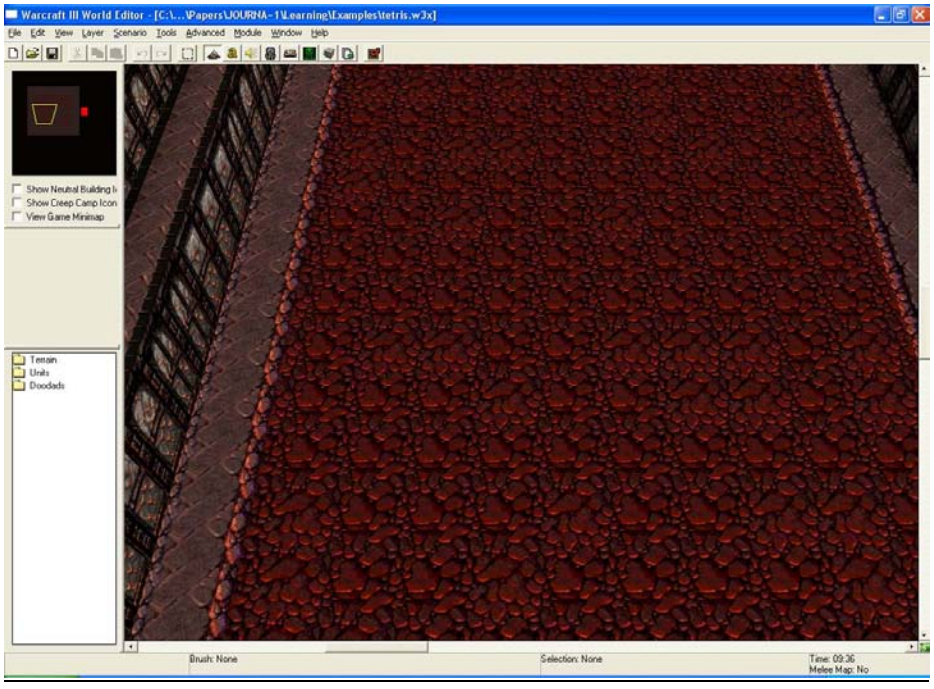

Fig. 5. The level design for the WarCraft III version of Tetris.

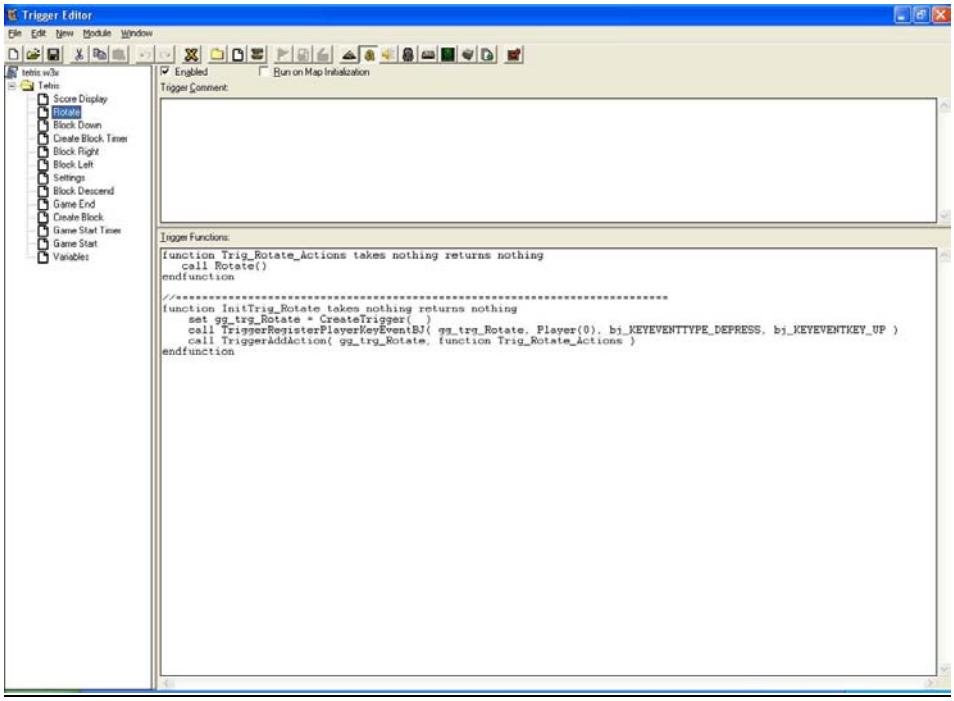

Fig. 6. Functions created by students for the Tetris game.

design, including Marc LeBlanc's MDA (Mechanics, Dynamics, and Aesthetics) theory [Hunicke et al. 2004]. We then discussed the game design process. Students were warned about the time needed for refining the design, and they took this consideration into account in their plan. Our critiquing sessions extended over two days, which allowed some groups to go over three to four iterations of their designs. Through this process, we believe they learned, on a small scale, the design and development cycle and the benefits of iterative design. We also observed that they were able to use and apply the MDA framework. Even though we didn't ask them to critique each other's games, they volunteered to do so.

Students were able to use the tools and quickly construct games in three days. They iteratively revised their games based on critiques and were able to understand the relationships between game mechanics and aesthetics. Since this was a group project, they also learned how to work in a group and divide the work evenly among group members, taking skills and the different tasks into account. They also learned several programming constructs, including threading, event-based programming, and rule-based programming. The Tetris group went a step further, writing their own functions to compute graphical rotations and translations.

The project encouraged them to work in groups of two. They were asked to develop a concept design, conceptualize the tasks involved, and divide them among team members. They were asked to fit their schedules within two, six-hour class periods. Such deadlines are unrealistic by professional game development standards where projects often take months or years to complete. We were surprised to see that students were able to keep their schedules and submitted their prototypes for review in the time allotted. We were

also pleased to see that the teams worked outside of class and used instant messaging and e-mail to communicate about their progress. Although this was a small project with a very small team size, we believe that they were able to engage in teamwork and were successful in developing a good project plan.

SECOND SETTING – GAME DESIGN CLASS

Classroom Setting

This second case study took place in a game design/programming class taught at The Pennsylvania State University. It was an advanced undergraduate class where 35 students from the Department of Computer Science and Engineering and School of Information Sciences and Technology worked together to design and develop a game of their choice. Since these students were more familiar with programming, we were able to introduce more complex game engines into the curriculum.

Process

The first part of the course introduced two engines: Wildtangent's Web Driver and Unreal Tournament 2003. For their first assignment, students were asked to build a small prototype to move characters around using the Web Driver library and C# as the programming language. For their second assignment, they were asked to make up a simple game modification to Unreal Tournament's DeathMatch game using Unreal Edit (map editor) and Unreal Script (scripting language, similar to java). These assignments were designed to help students become familiar with the engines before starting the process of creating their own games through their final projects.

During the second half of the course, students worked in groups of five to develop a game of their choice using any of the engines they learned in class. Note that the Web Driver is an engine that allows students to build their own games from scratch, thus it gives them flexibility and most of the functionality they need. On the other hand, Unreal is a game modding environment, where students implement their games by modifying existing game types, and thus they can reuse many of the functionalities already implemented in the Unreal games.

Web Driver Exercise. The Web Driver engine is a library composed of several classes that students can use to add 3D objects, maneuver them in 3D space, change camera view, etc. In their first assignment, we wanted to expose students to the main Web Driver functionality, such as animation, adding and maneuvering objects and characters, without overwhelming them with a very complex assignment. Therefore, we asked them to build a 3D environment with a character that receives stage directions, in the form of button clicks, dictating how it should behave.

To complete this assignment, students need to accomplish the following:

- Build a 3D environment using the Web Driver level editor, WTStudio. This entails understanding 3D coordinates. It requires them to place walls, objects, make textures and apply them on walls and objects, setup lights in terms of placement, angle and direction, and finally export that level into code.
- Add a character model (provided by the teaching staff) to the 3D environment using the Web Driver classes (specifically understand and use the WTActor class from the Web Driver Library).
- Given a destination position, figure out the forward orientation vector of the character, which requires applying dot and cross product.

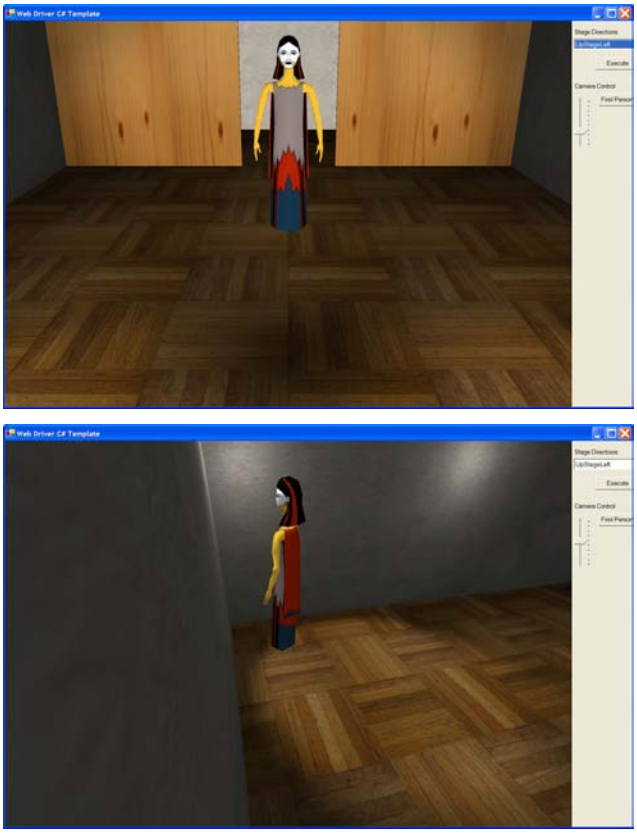

Fig. 7. An example Web Driver student assignment.

- Given a destination position and a forward vector, animate the character to move to the destination in a natural way, which requires knowledge of threading and event based programming.

We use a student project to demonstrate the complexity of the assignment and the content learned as a result. This example includes 865 lines of code written in C#, the code uses the Web Driver libraries. Figure 7 shows two screenshots of the example assignment. The opening scene is on the left, showing the environment created by the student with the correct textures and an aesthetically pleasing light setup. The screenshot on the right shows the scene after executing the command asking the character to move to "upstage left". The camera was programmed to move with the character, locking it in the center of view at all times (using the WTCamera class from the Web Driver Library). To get from the screenshot on the left to the one on the right, the character was programmed to rotate towards the destination (upstage left) then to move using a walk cycle animation. Using Web Driver classes, the student plays the walk cycle animation and moves the character a specific distance on a time interval. Manipulation of position and direction requires 3D math and simple vector geometry. The process of fixing the animation, distance, and timing to appear natural requires careful synchronization, threading, and event-based programming.

The following snippet of code was used to determine turning direction and to turn the actor:

```
IWTVector3D tempActorPosition = actor.getPosition();
WildVector ActorPosition = new WildVector(tempActorPosition.getX(),
tempActorPosition.getY(), tempActorPosition.getZ());

IWTVector3D tempOrientation = actor.getOrientationVector();
WildVector Orientation = new WildVector(tempOrientation.getX()*-
1,tempOrientation.getY(),tempOrientation.getZ()*-1);

WildVector GoalOrientation = new WildVector(GoalLocation.X -
ActorPosition.X, GoalLocation.Y - ActorPosition.Y, GoalLocation.Z -
ActorPosition.Z);

Orientation.Normalize();
GoalOrientation.Normalize();

float angle = GoalOrientation.TurnAngleXZ(Orientation);

actor.setRotation(0,1,0,angle,1);
```

All students wrote similar code, and only one of the 35 students was unable to complete the exercise. Many students attempted the extra credit part of the assignment, which required handling mouse events and adding collision detection using math and vector geometry techniques.

As we talked to students during this exercise, we noticed that many of them were unfamiliar with Pythagorean theorem or its relevance to game programming. The theorem is important because it is used to calculate the distance between two points in space. Since students were required to have a character move from one point to another, they needed to calculate the distance that the character had to travel. We also found that students had little knowledge of basic geometry and vectors. The instructor provided a lecture to help them understand these concepts. After the lecture, students reported that they understood the content and could display their knowledge when solving simple math equations provided in class exercises. However, they still had difficulties moving from solving equations to applying these concepts to 3D game programming.

Working with the Web Driver allowed them to apply and gain a better understanding of 3D geometry and vector mathematics. In particular, they applied the Pythagorean theorem and were able to calculate and use tangents, vector geometry (dot product), 3D transformations, and 3D rotations. These were difficult concepts to understand. We believe that by applying them in visual 3D space, students were able to visualize and assimilate their use and application.

In addition to learning basic math and geometry, students had to understand the use of object- oriented programming and component-based development. Most of the students had only taken a single object-oriented design course before this one, and they explained that they did not fully understand the advantages of object-oriented programming. Through this exercise, they were able to apply object-oriented concepts during the Web Driver exercise, using its libraries and component structure to successfully animate character movements.

Furthermore, any game development involves threading and event-based programming. In this particular assignment, students were required to handle mouse, keyboard, and render events as well as use threading for synchronizing character animation. Web Driver facilitates the use of event-based programming by supplying

several methods to trap events, such as keyboard, mouse, and rendering events. Web Driver functions are then used to handle these events. This concept was hard for students to understand. We gave them a quick tutorial using an example. Using this tutorial as a base, they were able to understand and develop their own events and event-handling mechanisms.

Unreal Tournament Exercise. The second assignment required students to modify the game behavior of Unreal Tournament by developing their own level and adding some Heads-Up Display (HUD) features to the Unreal's Deathmatch game. The assignment required students to create a level of two or more rooms using the Unreal Edit tool. It also required them to add a radar texture using the scripting language and display characters on the radar, depending on their position and orientation relative to the player.

The Unreal engine, unlike the Web Driver engine, requires modders to implement their modifications by extending functionality of existing engine classes via Class

Fig. 8. An Unreal Assignment created by a student.

Inheritance. This particular assignment requires that they understand (1) how to modify the code for the game using Object Oriented Design concepts, e.g. inheritance; (2) how to get positions of all characters in the level using the game classes; (3) how to calculate positions and orientations of different characters relative to the player; and (4) how to map this information on the radar. They were also required to color the blips on the radar depending on how dangerous the character is to the player. This requires that they access the internal state of each character to calculate danger in terms of the number of people they killed, their current weapons, etc. In addition, they were also required to indicate severe danger to the player via auditory or visual alarms. This required them to research different methods of displaying information through audio and/or the HUD.

All 35 submitted assignments met our requirements and were fully functional. Figure 8 shows screenshots from one of the assignments. The figures show the level design. Creating this level required them to understand 3D architectural design, texturing, lighting setup, and character navigation, since they had to lay out navigation points in their levels for the embedded Unreal navigation system.

The radar map shown in Figure 8 required students to create a radar texture and display it on the HUD using Unreal Script. They also needed to figure out how to reference all characters in the level, get their positions, and map their positions on the radar. This required them to understand Unreal's class structure and some simple math. A piece of the code from the example assignment displayed in Figure 8 is shown below to suggest the depth of programming knowledge required to complete this task. This particular piece shows how the student iterated through all characters within the level, drawing them on the radar based on their distance and orientation in relation to the player. As can be seen from the code below, this involves understanding the character class, called Pawn class used in the Unreal engine, as well as some math skills to map the distance of the character from the player into the radar.

```
foreach RadiusActors(class'Pawn', testpawn, 1024,PawnOwner.location)
{
  if(testpawn != pawnOwner)  //don't check agains the owner
  {
    magnitude = vsize(testpawn.Location-PawnOwner.Location)/8;
    newRotation =rotator(testpawn.Location-PawnOwner.Location);
    correction.yaw = 65535/4;
    newRotation.yaw -= (PawnOwner.Rotation.yaw +correction.yaw);
    newPosition=vector(newRotation)*magnitude;
    //set the draw position & scale it
    c.SetPos(radarscale*(newPosition.x + Radar_Tex.USize/2) +
radarTL_X,radarscale*(newPosition.y + Radar_Tex.VSize/2) + radarTL_Y );
    //place the blip on the radar
    c.DrawIcon(getBlipTexture(c, testpawn),radarScale);
  }
 }
```

We noticed that students were much more motivated to finish and perfect this assignment than the Web Driver assignment. They all attempted the extra credit code part of this assignment, as compared to only around 40% who attempted the extra credit for the Web Driver assignment.

While the Web Driver assignment stressed 3D geometry and vector math, using the Unreal Tournament exercise required students to understand architecture design, texturing, sound design, lighting design, and landscaping, in addition to basic understanding of 3D geometry in order to use the 3D modeling tools. Additionally,

students were able to gain a better understanding of event-based programming and threading than with the Web Driver. This is understandable, due to the fact that the Unreal engine is much more complex with many threads and events working in parallel. In order to work in such an environment, students needed to understand the complex structure of the engine before they completed their assignments. As can be seen from the code above, students needed to explore character information and to figure out how to iterate through all characters in a level. These are example problems that require students to explore the structure of the engine including the classes and methods that can be used to get the job done.

The engine itself uses an event-based architecture where all systems within the engine communicate through events. An understanding of this basic structure is important because only such understanding will enable the students to solve the assignment. We observed that learning the engine's event-based structure was hard for them at first, but many of them struggled through it. Those who collaborated and were persistent enough found the solution. On work days (lab classes), we observed that students were very engaged in developing their own modifications to the game. When they hit a problem, they tried to find a solution themselves. If they couldn't, they asked each other until they found an answer.

Also due to the complexity of the Unreal engine, we observed that students critiqued the engine code supplied through the scripting language. They made many comments on the use of global variables and the optimizations used in the engine. This promoted a better understanding and appreciation for good coding standards.

The Unreal engine also promoted the use of inheritance and object-oriented programming. Through this assignment, students understood the value of reuse and inheritance, which was surprising to us since these topics were discussed before in object-oriented design and software engineering classes. We deduced that students didn't understand the implication of these concepts until they applied them in a complex and relatively large-scale project.

In addition to these concepts, we observed that by building their own 3D architectures students learned architectural design, lighting, texturing, and limitations of real-time rendering and effects. By adding characters to the environment, the students learned how to use the built-in algorithms for navigation and make their characters navigate through their architectures.

STUDENT PROJECTS

Through their projects the students gained a much greater understanding and assimilation of the concepts discussed above. In addition, they also learned project management, scheduling, and iterative development. Additionally, concepts of game design were better promoted through their projects. Students iteratively built their games through critiques. This allowed them to reflect on and adapt several game design theories that were discussed in class.

Movies of some of the student projects can be found at http://courses.ist.psu.edu/SP05/IST402/ projects.html. We discuss one particular example to suggest the complexity of these projects. The game was based in 13th century Scotland, the player character is a 13-year old girl who finds herself imprisoned in a dungeon. She struggles to get out of the dungeon and avenge her imprisonment. There were no guns in the 13th century. Since the students used the Unreal engine and the Unreal engine is based on a first-person shooter game, they had to alter the engine and create their own set of weapons, weapons that are more appropriate for the era chosen for the game.

Their final project (http://courses.ist.psu.edu/SP05/IST402/Projects/Student Finals/DungeonGirl. wmv) included over 6020 lines of code, 5 characters, 24 weapon models, 5 original music files, and 80 sound effects and dialogue files. In addition, three complete levels were created for the game, including lighting design, textures, and navigation points. Students had to complete a number of tasks, including the following:

- Create 3D models of skeleton characters, swords, bows, and arrows.
- Create 3D levels; 3 main levels were created for this game.
- Adjust the game code to enable melee-type engagement, this requires:
    - adding new weapons and removing the guns;
    - creating and working out the animation for bow and arrow movement;
    - adding distance measurement for sword impact;
    - adding scoring mechanisms for damage and impact behavior; and
    - disabling guntype weapons normally used in Unreal.
- Adjust game code to include cut scenes at appropriate moments.
- Create dialogue and script.
- Record voice acting and composing music.
- Add sounds and music.
- Adjust game code to include the voice acting, keeping track of story line, adding sound effects.
- Adjust game code to calculate winning and losing conditions, transitions between levels, etc.

Building a game using Unreal and/or Web Driver engaged students at many levels and provided a good learning environment, where they learned many concepts related to programming, object-oriented design and development, 3D geometry, 3D vector math, navigation, and elementary artificial intelligence. They also learned coding standards and quality, as well as project management, iterative design, 3D modeling, architecture design, and art content development.

OBSERVATIONS

In this section we discuss the concepts we observed students learn during this class. We believe that using both the Web Driver and Unreal Tournament engines promoted understanding and assimilation of different concepts. The Web Driver promoted learning of programming and 3D math concepts, such as 3D geometry, rotations and translations in 3D space, object-oriented programming, and component-based design and implementation. On the other hand, the Unreal engine promoted learning of threading and event-based programming, architecture design, 3D modeling, and object-oriented programming and reuse.

Their project work promoted these concepts further, and also an understanding of other topics including software design, iterative design and prototyping, project management, team work, and project scheduling.

At the end of the semester, we asked students to describe what they learned from the coursework. Table I shows some of their responses. The majority of the comments dealt with general issues of game design, but there were also some discussions of software engineering and programming concepts.

We also asked student to rate the Web Driver and Unreal exercises in terms of their interest and difficulty (Tables II and III). The perceived difficulties for the two assignments are similar (46.5% for Web Driver vs. 53.9% for Unreal Tournament on the

Table I. Student Reports on Concepts In Game Programming Class

Comments on Design Process

1. The role of narrative in game design. Most excellent games tell a story. This is crucial in building a compelling experience for the user.
2. Gamers do not make designers. It takes thought, reflection, testing, and imagination; something that many gamers lack: hence game designers.
3. The idea that design is more important than anything else: Without good structure/gameplay/engines, an idea, no matter how "cool" cannot save a game.
4. How to make a game more immersive and how to avoid the "pitfalls."
5. The importance of characters and story-telling.
6. That there are many aspects to game design that must be considered. Also there must be a separation between game design and game programming. Implementation is an entirely different realm than design concepts.
7. Various aspects of game design.
8. Overall game design.
9. High interactivity.
10. Game design concepts in general. It forced me to focus on the design principles of a game instead of just deciding whether it was "fun" or not. I definitely have even more appreciation for all the work that goes into designing a game.
11. Good game aspects. Utilizing what has and hasn't worked in the past will allow me to make a better game for the future.
12. Making the experience immersive, i.e., sound, light, everything. I also have a much greater respect for the difficult process of game design/creation.
13. There's a lot of stuff going on in games that you really don't think about while playing, but were big considerations during the production of the game.
14. Interactive narratives. This seemed to be a big topic for many people when talking about games in forums and at other times during class.

Comments on Software Engineering-Based Concepts

15. Coding and style of coding because nothing works if you can't code correctly.
16. I learned that tools are very useful and you can find a tool to do anything.
17. How to program and produce something in a group.
18. It's not necessarily a concept, but I learned how to write UTScript. I had very little experience in OO programming prior to this class, and my work with UTScript helped me not only understand OO concepts, but also allowed me to learn a way to customize a game engine.

General Comments

19. Game programming is not for me... Too complicated.
20. Learning to create mods and mutators, because I really felt it helps to understand the different components of a game. It's also a great experience to have been able to spend time creating a level and a mod in itself.
21. Unreal Scipt is a pain in the behind.

hard and very hard responses), but the Unreal Tournament exercise was rated more interesting than the Web Driver assignment (46.2% for Web Driver vs. 88.4% for Unreal Tournament on the interesting and very interesting responses). Their interest in Unreal

Table II. Difficulty Ratings for the Web Driver and Unreal Tournament Assignments

Difficulty ($N = 26$)	Very Easy	Easy	Average	Hard	Very Hard
Web Driver	0 (0.0%)	4 (15.4%)	10 (38.5%)	9 (34.6%)	3 (11.5%)
Unreal Tournament	1 (3.8%)	0 (0%)	11 (42.3%)	10 (38.5%)	4 (15.4%)

Table III. Interest Ratings for the Web Driver and Unreal Tournament Assignments

Interest ($N = 26$)	Very Boring	Boring	Average	Interesting	Very Interesting
Web Driver	1 (3.8%)	3 (11.5%)	10 (38.5%)	4 (15.4%)	8 (30.8%)
Unreal Tournament	0 (0%)	1 (3.8%)	2 (7.7%)	7 (26.9%)	16 (61.5%)

Tournament was also apparent in the course project: The majority of students chose it over Web Driver as their game engine for the project. This may be due to the fact that Unreal is a complex game that is similar to the games that students normally play. It may also be due to the fact that most students have played Unreal Tournament before, which provides them with a familiar environment that they can customize and personalize.

DISCUSSION

Through these two classes, we gained better insight on the use of game modding as a tool to promote learning. We believe that there are several skills and concepts that students learn by engaging in game design/modification, including the following:

- *Software Development and Design*
  - team work
  - building critiques and reflections on other's work
  - project scheduling
  - project management
  - iterations and refinement
  - prototyping
- *Programming Concepts*
  - threading and event-based programming
  - Object-oriented programming
  - Component-based development
  - Software patterns
- *Artistic Concepts*
  - Lighting
  - Architecture design
  - Character design
- *Game Concepts*
  - Game design
  - Game mechanics
  - Balancing game aesthetics and game play

All of these concepts could be taught in isolation, but having students apply them in design tasks will help them develop more holistic views of software engineering

practices. Game engines provide foundations for design exercises in the form of tools, libraries, and prebuilt objects. These let students quickly experiment with design concepts by modifying existing game worlds. This may be important for student motivation.

For instance, students found the Unreal exercise more engaging than Web Driver, and they seemed to work harder at understanding the engine and acquiring the skills needed to complete their games. This may be due to that fact that students have come to appreciate and expect the same aesthetic quality that they see in games today when working with games in an educational setting. Students told us that they gravitated towards Unreal because it gave them a complex environment to manipulate while preserving the aesthetics that they were accustomed to in their own game playing. Prior experience playing Unreal may also provide some motivation. Since they are familiar with the game, it may be more interesting to develop projects that resemble it as well as being able to share a familiar product with friends and peers.

It is not enough to give students arbitrary game engines and expect them to learn computing skills. We chose the three game engines in our courses because we believed they could assist in the development of particular skills. For instance, WarCraft III was used with high school students because they had little programming experience. Its visual tools allowed students to create working games and to explore computing concepts without getting wrapped up in the syntax of programming languages. In particular, WarCraft III promoted learning the following:

- programming basics, including Boolean logic, conditionals, and variables;
- threading and parallel programming;
- simple concepts related to landscaping, modeling, animation, and camera motion.

On the other hand, the Web Driver was chosen to explore its flexibility and its use to promote 3D vector geometry and other graphics concepts. According to our observations, we see the Web Driver promoting the following skills and concepts:

- 3D vector geometry and math concepts, including dot and cross product calculation, tangent calculations, the Pythagorean theorem, and calculating rotations and translations in 3D space;
- animation and synchronization of different actions for a character, e.g., rotations and animations
- using libraries and object-oriented programming;
- simple 3D level design, including lighting, creation, and manipulation of textures;
- event-based programming, triggering and handling of input events (keyboard and mouse events), as well as engine-based rendering events;
- threading and thread synchronization.

Unreal Tournament was chosen because it is a complex game architecture that has a great deal of game functionality embedded in its engine. It was also used to target the following skills and concepts:

- 3D architecture design, including lighting and creating, and manipulating textures; adding sound effects and music to create the ambience of 3D space;

- eventbased programming using Unreal's event;
- threading and synchronization;
- 3D modeling;
- navigation systems;
- objected-oriented programming, specifically inheritance and reuse.

CONCLUSION

In conclusion, in this article we have presented some evidence that encourages the use of game modding in classes to promote learning of several subjects and concepts. We discussed the concepts we believe students learned through the cases presented here. Our preliminary evaluations of learning were based on student performance and our observation and interactions with them through their assignments and class discussions. We believe that using game modding motivated students to learn and allowed them to apply and visualize the utility and application of the concepts. We also observed that different game engines implicitly stress the use and development of certain skills. This becomes an important issue when choosing engines for pedagogical purposes.

ACKNOWLEDGMENTS

We would like to acknowledge all the students who participated in the two classes for making them such a success. We also would like to acknowledge our teaching assistants, Ibrahim Yucel and Joseph Zupko, who made this class so easy to teach and helped promote a better learning environment for the students.

REFERENCES

ANGIOLILLO, P. 2005. Gaming making the grade. *Technology Review* (Sept 27, 2005).

CONWAY, M., AUDIA, S., BURNETTE, T., COSGROVE, D., CHRISTENSEN, K., AND DELINE, R. 2000. Alice: Lessons learned from building a 3d system for novices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. T. Turner and G. Szwillus (eds.). ACM Press, New York:

EMMERSON, F. 2004. Exploring the video game as a learning tool. *ERCIM News 57*, 30 (2004).

HAREL, I. 1991. *Children Designers: Interdisciplinary Constructions for Learning and Knowing Mathematics in a Computer-Rich School*Ablex Publishing, Norwood, NJ.

HOOPER, P. K. 1998. *They Have Their Own Thoughts: Children's Learning of Computational Ideas from a Cultural Constructionist Perspective*. Cambridge, MA.

HUNICKE, R., BLANC, M. L., AND ZUBEK, R. 2004. MDA framework for game design. In *Proceedings of the Game AI Workshop* (San Jose, CA, 2004). AAAI.

HYMAN, P. 2004. Video game companies encourage "modders". *The Hollywood Reporter*.

INGALLS, D., KAEHLER, T., MALONEY, J., WALLACE, S., AND KAY, A. 1997. Back to the future: The story of squeak, a practical smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York.

KAFAI, Y. 1994. *Minds in Play: Computer Game Design as a Context for Children's Learning.* Erlbaum.

LEHRER, R. 1986. Logo as a strategy for developing thinking. *Educational Psychologist 21*, 1/2 (1986), 121-137.

MOSKAL, B., LURIE, D., AND COOPER, S. 2004. Evaluating the effectiveness of a new instructional approach. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education.* D. Joyce and D. Knox (eds.). ACM Press, New York, 75-79.

PAPERT, S. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York.

PEA, R. D., KURLAND, D. M., AND HAWKINS, J. 1987. Logo and the development of thinking skills. In *Mirrors of Mind: Patterns of Experience in Educational Computing*. R. D. Pea and K. Sheingold (eds.), Ablex Publishing, Norwood, NJ.

PUNTAMBEKAR, S. AND KOLODNER, J. L. 2005. Toward implementing distributed scaffolding: Helping students learn from design. *Journal of Research in Science Teaching 42*, 2 (2005), 185-217.

REPENNING, A. AND AMBACH, J. 1997. The agentsheets beahvior exchange: Supporting social beahvior processing. In *Proceedings of the CHI 97 Conference* (New York).

RESNICK, M. 1994. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds.* The MIT Press, Cambridge, MA.

RESNICK, M AND OCKO, S. 1993. *Lego/logo: Learning Through and About Design*. Ablex Publishing, Norwood, NJ.

SMITH, D., CYPHER, A., AND SPOHRER, J. 1994. Kidsim: Programming agents without a programming language. *Communications of the ACM* (1994), 54-67.

SMITH, D., CYPHER, A., AND TESLER, L. 2000. Novice programming comes of age. *Communications of the ACM 43,* 3 (2000), 75-81.

WRIGHT, W. 2004. Triangulation: A schizophrenic approach to game design. In *Proceedings of the Game Developers Conference* (San Jose, CA).

Received August 2005; revised October 2005; accepted August 2005