Teaching Graphics for Games using Microsoft XNA

Radu Paul Mihail, Judy Goldsmith, Nathan Jacobs and Jerzy Jaromczyk
Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506
Email: r.p.mihail@uky.edu, goldsmit@cs.uky.edu, jacobs@cs.uky.edu, jurek@cs.uky.edu

***Abstract*—We present an approach to teach computer graphics for game development using Microsoft XNA as a development platform. Our student body consisted of undergraduate students who just completed CS2 to seniors in their last semester. Our course had 4 individual programming assignments that we crafted in a way to encourage easy assimilation of mathematical concepts as well as to make debugging easy through visual feedback. We discuss the topics we covered, pedagogical methods used and student feedback for each of the programming assignments.**

I. Introduction

Goals of game programming courses include teaching programming, software engineering, algorithms and computer graphics. In this paper we present an approach to teaching computer graphics and the accompanying mathematics in the context of a game programming course using XNA. We discuss the assignments used in our course and observations related to students' comprehension of the mathematical techniques required to complete them. Our students ranged from those who just completed CS2 the previous semester to seniors in their last semester before graduation. We claim that our course was successful in meeting the listed goals through our carefully crafted assignments and the use of on-line discussion forums as a peer support tool. We discuss our observations about the effectiveness of visual feedback for learning and applying mathematical tools used in computer graphics in the context of a game programming course.

A. Gaming in CS Education

Many CS educators try to make course material more appealing in an effort to attract more students. One of the approaches is to add a "fun factor" to courses such as software engineering or algorithms. Games provide a platform for teaching concepts such as object oriented programming, algorithms, computer graphics. Given the increasing popularity of games, many students dream of developing their own. However, complicating factors exist, such as the increasing complexity of modern games from the myriad of available techniques coupled with the mathematical tools required. Efforts to enhance courses by making them "game-centric" have been shown to be successful [1] where student success is measured by student satisfaction surveys, grades and teacher course evaluations (TCEs). In this paper, we describe a course taught in theComputer Science Department at the University of Kentucky titled "XNA Game Programming". We taught graphics programming using XNA as a platform in a course where groups of students produce a working game as a final project. We discuss the general structure of the course and observations regarding students' performance on programming assignments. We report observations on student results and discuss how visual feedback can enhance the learner experience for certain simple game assignments with specific learning outcomes. First, we give a brief overview of existing work in CS education where gaming technology is used as a motivating factor throughout a curriculum.

a) Programming: Debates on how to teach CS1 are as old as academic computer science. A recent trend, most likely sparked by dropping enrollment rates, employs introductory programming using a game creation theme. Leutenegger et al. [1] used Flash/ActionScript as a first language due to the low initial learning overhead. They reported increased enrollment and retention rates. Moreover, they showed that, despite claims that a game-focus would discourage women, retention for men and women was equal. Kölling et al. [2] employed a similar approach to Chen et al. [3] and incorporated gaming into object oriented programming to motivate students. Both Kölling and Chen [2], [3] report success due to "immediate and intuitive feedback about program behaviour",[2] but give no further details. Our class was aimed at students who have already taken CS2 and were expected to have basic programming competence and familiarity with an object oriented programming language.

b) Algorithms and Data Structures: Attempts to make abstract programming concepts and algorithms more attractive can also make use of games. Tan et al. [4] published a case study where second year IT students are taught data structures using a game that helps them visualize data structures and access operations on stacks and queues. Hakulinen et al. [5] use card games as teaching tools for algorithms. While a comprehensive review of the related literature is beyond the scope of this paper, we emphasize that more work needs to be done in quantitatively evaluating the effectiveness of "game-centric" courses with respect to specific learning outcomes.

c) Software Engineering: Wang et al. [6] describe a case study where they integrated game development using XNA into a software engineering course. Here, game development was used in an attempt to motivate and increase interest in the final project. They reported that students had a more positive attitude toward the game focused course, as opposed to the standard course, although some students spent more time on developing the game than on the software architecture. It is worth noting that the authors in [6] mentioned a sparse literature on architecture in the game development domain. The reader can refer to [7], [8], [9] for other examples.

d) Computer Graphics: Computer graphics is a large part of the game development process. Standard graphics classes focus on mathematics and algorithms. Du et al. [10]

claim that the traditional methods of teaching computer graphics to students whose focus is game development is ineffective due to poor mathematical foundation. In our course, we attempted to bridge this gap by adding a "fun factor" to the mathematics required for graphics in game development. We introduced the required mathematics and constructed assignments such that visual feedback during the implementation phase served as a troubleshooting tool and discouraged shallow understanding of the underlying mathematics. Feedback from students' write-ups seem to support our hypothesis that carefully crafted assignments lead to better assimilation of the materials taught.

II. XNA IN THE CLASSROOM

XNA is a collection of tools from Microsoft designed to ease the burden on game programmers by offering built in functionality commonly used in games. Examples are a rich content processor, classes that implement mathematical functions often used in graphics programming and an easy to use API for sound and music playback. This burden release offers educators a tool that can be integrated in technical programs. In one such example, Linhoff et al. [11] discuss GAM 380, a course offered at the School of Computer Science, Telecommunications, and Information Systems at DePaul University (DePaul CTI) in their Game Development Program. The intent of this course was to focus on content creation in a mixed programmer and non-programmer student population. Students were given sample programs for which they would create fonts, icons, 3D models animations and other content. They were encouraged to extend the samples, although programming was not taught as part of the class. They report positive student responses, and note that grades for students with a richer programming background were on average a letter grade higher. This finding suggests that mixing artists and programmers can be done, but more work needs to be done to ensure skill sets are complementary. Our course was strictly technical with a focus on graphics programming for games.

A. *Material covered in our course*

This class was intended to give students an exposure to basic concepts and techniques for game programming. We started by introducing XNA and C#, the game loop, game states, scoring, polled input, game components and emphasized the importance of OOP. Our next goal was to lay the foundations of 2D game graphics through the use of sprites and primitives. Drawing sprites in XNA is straightforward: SpriteBatch.Draw() method has several overloads, all of which have a position parameter (Vector2D) in *screen coordinates*. Drawing primitives in XNA is not as straightforward; it requires understanding the full graphics pipeline. We avoided prematurely teaching this by giving students "helper" code which implemented orthographic projection (covered later in the course), to facilitate drawing primitives (e.g., lines and triangles) in screen space. We then introduced vector and matrix operations that were applied to derive transformations in 2D (rotations, translations and scaling) and demonstrated during lecture. This allowed a smooth transition to 3D graphics and a presentation of the full graphics pipeline, although students were tested on a small subset of the concepts presented (those required for the programming assignments). Projective geometry was the next topic covered; we shed light on the previously mysterious "helper" code (orthographic projection) and derived the perspective projection matrix. Given this foundation, we then taught texturing, 3D models, cameras and collision detection. The "mystery" thus far was the use of XNA's BasicEffect, until we covered High-Level Shader Language (HLSL). The course concluded with keyframe and skeletal animation.

B. *Pedagogical methods used*

Our course had a lecture component, 4 individual programming assignments, 5 quizzes, weekly on-line discussion forums and a final group project. During the lectures, mathematics were formally introduced and applied in the XNA framework to demonstrate results. The programming assignments were small games. Students had to implement the techniques learned in class and produce a working game for each of the four assignments. Our course also employed online discussion forums intended as a peer support tool. Students had to post an original thread every week (minimum of two paragraphs) and respond to three posts from other students (a minimum of one paragraph). We found this to be an effective means of getting peer support. We are verifying this finding.

C. *Course objectives*

The principal objective was to provide students basic knowledge of game development, with a focus on computer graphics. The prerequisite for this course was CS2, which meant we could not assume a strong mathematical background. We planned to introduce concepts from linear algebra and algorithms as needed through the progression of the course and tailored the programming assignments in a way that encouraged discovery and motivate the assimilation of concepts through final products: simple games.

In the beginning of the course we administered a survey to elicit prior knowledge of the topics to be covered. Below is a summary of relevant questions:

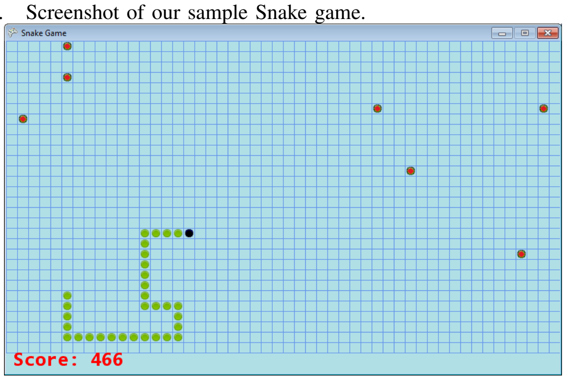
- Have you taken a course in linear algebra?
  Yes: 58%, No: 42%
- How confident are you in your programming skills (object oriented programming in a high level language)?
  Very confident: 22%, Confident: 58%, Not confident: 9.6%, Insecure: 9.6%
- Are you familiar with the pinhole camera model?
  Yes: 32%, No: 67%

More than half of the students reported taking linear algebra prior to this course. Although linear algebra was not required, we note from our observations that it helps to greatly reduce anxiety about most topics in a game programming course with a focus on computer graphics.

III. PROGRAMMING ASSIGNMENTS

We now present the programming assignments used in this course. Three of the four assignments were meant to be complete working games (albeit simple) with goals and

Fig. 1. Screenshot of our sample Snake game.



objectives defined by the students. This freedom to choose the game objectives was appreciated; they reported putting more effort into the projects.

A. *Snake Game*

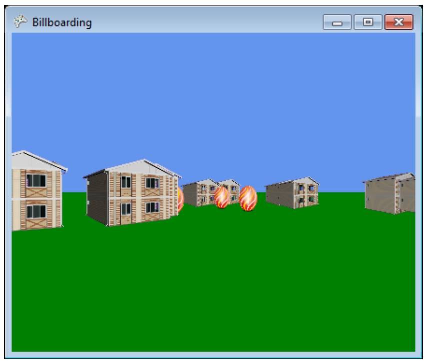
The first programming assignment in this course was a simple snake game (see Figure 1). As a first contact with the XNA framework, this project encouraged the discovery of solutions to typical problems in 2D game programming. Drawing sprites in XNA is straightforward; XNA's *SpriteBatch.Draw* method provides a variety of options, including scaling, transparency handling and rotation among others. For extra credit, students were given the freedom to improve the final appearance of the game. For example, the sprite representing the head of the snake can change as a function of direction. Another improvement is to use include animations (spritesheets) to make the appearance more pleasing.

The XNA framework includes a game loop with two important functions: Update() and Draw(), both of which are called at a default rate of 60/second. Most snake implementations are grid based; the snake is usually composed of discrete blocks that advance in the current direction at a rate much lower than 60 frames/second. Students had to reduce the frame rate in one of two ways: changing the *TargetElapsedTime* of the *Game* class, or advancing the snake once every x frames. This decision had a direct impact on the responsiveness of the game because of the way input is handled through polling. Changing the rate at which Update() and Draw() are called means a short-lived keystroke may not be detected, while constraining snake movement every x frames and polling every frame increases responsiveness. The majority of students picked the most responsive option, despite the small increase in code complexity.

The snake can collide with itself, screen (window) boundaries and "food". Contingent on students' decision of how to store the world state (grid constraints or screen pixels), collisions can be more or less accurate. Here, students were encouraged to discover the trade-offs of these two approaches.

Student feedback A common source of confusion stemmed from drawing grid lines. When the client window is resized, a common scaling factor needs to be applied to both primitives (lines) and sprites. Students seemed to have difficulties thinking of different world spaces coexisting simultaneously. Another commonly reported problem was how to structure code. We encouraged the use of best practice OOP, however, we left decisions such as what the functionality of a single *GameComponent* is to them. We did not require students to submit a design prior to submission; we will ask in the future.

Fig. 2. Screenshot of our sample billboarding game. In this game, the goal was to find all the "eggs" in a world filled with randomly placed houses drawn as billboards.

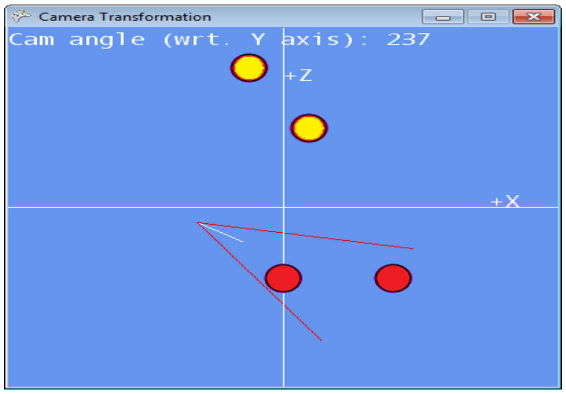


B. *Billboarding*

Transitioning from programming flat game worlds to a third dimension introduces several complicating factors: solid understanding of projective geometry, extending standard (scale, translation and rotation) transformations to 3D using homogeneous coordinates, and the abstractization of raster operations (i.e., thinking in terms of arbitrarily scaled spaces). Our pedagogical approach to this step of the learning process revolved around a programming assignment we uninterestingly titled *Billboarding*. We eased the transition to 3D by first introducing orthogonal projection. This is conceptually easy to grasp because in its simplest form, it involves simply discarding the z component of a point in 3D, thus *projecting* all points on the XY plane that can be mapped to screen coordinates. While students claimed to understand the concept of projection both as a many-to-one mapping from 3D to a plane and as a linear transformation, some were vocal about lacking intuition, i.e., "Why does it work?" Projection is a topic taught in linear algebra and requires defining vectors, matrices, vector spaces, range, null space of a matrix, etc. We used art as a teaching tool by showing examples from Renaissance artists when perspective started to be formally stated (e.g., Abrecht Dürer's "The Painter's Manual", ca. 1525). This encouraged lively classroom discussion, where students with a background in art could contribute. This discussion gave the instructor a smooth segue to the derivation of perspective projection rules using similar triangles. This was stated first as a point dependent matrix, later extended to a point independent projection matrix using homogeneous coordinates.

The billboarding assignment was intended to be a preamble to using the full graphics pipeline. The idea was to create a small game using a first person camera where the objective was to find some objects hidden in the world or some other interaction of students' choosing (see Figure 2). The world was restricted to a plane where the player and objects "live". We taught the concept of a camera as an "alignment tool": we know how to (easily) project points onto the XY plane, but the camera's image plane does not match the worlds', so

Fig. 3. Top-down view of the camera transformation. This program was given to students to help them visualize the camera transformation. Red objects are drawn in world space and the yellow objects are the same objects in camera space. The camera can be moved and rotated using the arrow keys.



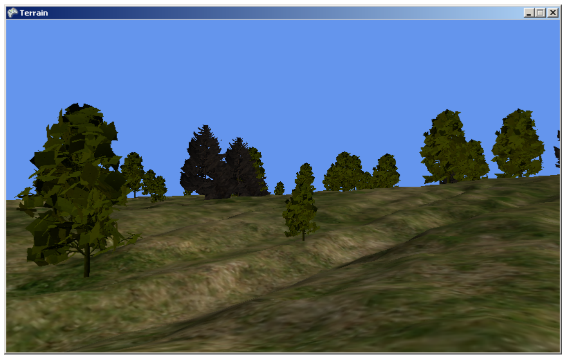
a rigid body transformation needs to be done to align them. The *projection* and *view* matrices can easily be constructed in XNA using the *Matrix.CreatePerspective* variants and *Matrix.CreateLookAt*. For this assignment, students were tasked with implementing their own pipeline, with a restriction to only use XNA's *SpriteBatch.Draw* method, where the position argument is in screen space.

We now discuss the steps involved in solving this problem and the difficulties students encountered. The first decision was the scale of the world. We encouraged students to restrict the world to a unit square, although any scale could be used. Next, students had to implement a camera, capable of translation (in the direction given by the *forward* and *side* vectors) and rotation (yaw, along the *up* vector, which is kept constant). This is simply a rigid body transformation in 2D that can be easily visualized. We provided an example to help students experiment with the camera transformation (see Figure 3). This example encouraged them to check the results of their computations and avoid common pitfalls such as performing the rotation prior to the translation (although some students did that).

The next step is to draw the objects using *SpriteBatch.Draw* calls, which requires computing the scale of the sprite with respect to the distance from the camera and a 2D projection (x') from the XZ plane to the image plane. The image plane was encouraged to be $z = 1$ (focal length d of 1), such that students would arrive at the simple result $x' = \frac{x}{z}$ using the same method based on similar triangles we used to derive the point dependent perspective projection matrix. This result needs to be scaled such that it matches screen coordinates for clipping to occur naturally (XNA's *SpriteBatch.Draw* behavior is quite tolerant of such abuses). Finally, the sprites had to change with respect to the angle between the object and the camera to achieve a 3D effect.

Student feedback The most common reported problem was difficulty understanding the mathematics required to implement the camera and projection transformations. The order of operations (translation/rotation) in the camera transformations provided visual feedback that led many students to fix the problem while others asked the instructor for help. Computing the angle between the camera and objects was also difficult for a small percentage of our students. The projection operation required scaling the sprites, which was also reported to cause difficulties in completing this project.

Fig. 4. Terrain based game. The students were free to choose goals/objectives of the game.



C. Terrain based game

The third assignment in our course was similar to the billboarding assignment, but the world was no longer flat (see Figure 4). Instead, students generated a terrain mesh from a height map and used the full graphics pipeline to draw the world while constraining the camera's y position to the surface of the terrain. This used bilinear interpolation so that movement was smooth, irrespective of the terrain "roughness".

The assignment was designed as an extension of the billboarding assignment. Before this project was due, we taught how to import, use and draw 3D models in XNA. XNA includes the *BasicEffect* class, a useful tool for someone without any HLSL experience. Students were encouraged to use the full functionality of the *BasicEffect* class that includes texturing, basic lighting and fog.

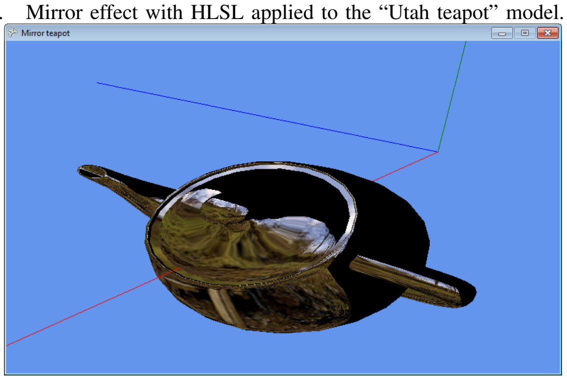
Collision detection was discussed throughout the course. The XNA framework provides beautifully interfaced collision detection algorithms for various primitives (e.g., bounding boxes and bounding spheres). To get full credit, students had to "plant" at least two models in the world and implement some form of interaction with the models. Most students went beyond the basic requirements and created simple first person shooters.

Design decisions prior to writing code proved imperative in successfully completing this assignment. A map from the camera's position on the XZ plane to a discrete array of elevations from a height map was needed. If this mapping was incorrect, the player would "walk" through the terrain and models would be placed incorrectly. Here, students could troubleshoot incorrect mapping by observing placement of the camera and models.

Student feedback A common problem students reported was related to the behavior of *SpriteBatch.Begin()* method. Although documented, this call[1] changes a number of device

[1]Most students chose to display game state variables (e.g., lives, time, etc.) using a *SpriteBatch.DrawText()* call.

Fig. 5. Mirror effect with HLSL applied to the "Utah teapot" model.



states: *BlendState, DepthStencilState, RasterizerState and SamplerStates*, among others, causing 3D objects to render incorrectly. Students with a solid foundation in graphics identified this problem easily, while others asked for help.

D. Mirror effect

We covered HLSL near the end of the semester. This topic was popular with students because it shed light on XNA's *BasicEffect* effect class, which many students thought of as a "black box" rich with functionality. Our approach to teaching HLSL was hands on: write code during lecture and discuss various aspects of the language. Students who had prior experience with graphics programming, but not with shader code, reported that seeing the instructor write HLSL code in class helped reduce anxiety tremendously. We started with texture mapping, followed by theory on diffuse and specular reflection models. We then demonstrated different shader models (per pixel or per vertex) and discussed tradeoffs.

Our goal with the fourth assignment in this course was to give students hands-on experience with HLSL to solve a problem for which the resulting solution can be intuitively verified for accuracy: a reflective shader. The assignment was set up such that an image in the world (on a plane parallel to the yz plane) would be reflected on a model rendered using a shader the students were asked to write. The choice of the image on the yz plane was to simplify sampling the texture map after the ray-plane intersection computation.

For this assignment, we provided a Visual Studio project, complete with all the C# code needed. We also provided two models (a cube and the famous Utah Teapot) and encouraged students to thoroughly test their solution with other models to ensure correctness. During runtime, the models would move and rotate in the world. The existence of a world transformation (with a translation component) required discarding the translation components of the world matrix when transforming the normal vectors passed to the shader from the model.

The solution to this assignment requires the following computations: incident vector ("eye" vector), reflection vector and the ray-plane intersection between the reflection vector and the plane on which the image is placed on. The ray-plane intersection computation involves solving for t such that the point on the plane is $p = p_0 + \mathbf{t} * ref\_vector$ where p_0 is a model vertex. Because the image to be reflected was on the yz plane, it sufficed to use $p.y$ and $p.z$ as arguments for the texture sampler call.

Student feedback This assignment was reportedly the most rewarding for our students. Few had prior experience with shaders, so the successful completion of this assignment increased their confidence and some groups implemented effects with HLSL for their final project. A common complaint was the inability to use a debugger such as the one offered by Visual Studio. Since the assignment involved relatively few computations, students could pin-point problems by running the program and visually inspecting the results.

IV. Conclusion

In this paper we presented an approach to teaching graphics for game programming using XNA. Our student body consisted of students who took CS2 the previous semester to seniors close to graduation. We discussed the materials we covered in this course as well as the programming assignments. The prerequisite for this course was CS2 (programming in an object oriented language). While linear algebra was not required, students who had taken that class were clearly more comfortable with the materials covered and completing the assignments, a fact reflected in the final grades. The programming assignments were crafted so that the assimilation of materials was rewarding (3 of the 4 were working games) and debugging was helped by visual feedback.

References

[1] S. Leutenegger and J. Edgington, "A games first approach to teaching introductory programming," in *ACM SIGCSE Bulletin*, vol. 39, no. 1. ACM, 2007, pp. 115–118.

[2] M. Kölling and P. Henriksen, "Game programming in introductory courses with direct state manipulation," *ACM SIGCSE Bulletin*, vol. 37, no. 3, pp. 59–63, 2005.

[3] W.-K. Chen and Y. C. Cheng, "Teaching object-oriented programming laboratory with computer game programming," *Education, IEEE Transactions on*, vol. 50, no. 3, pp. 197–203, 2007.

[4] B. Tan and J. L. K. Seng, "Game-based learning for data structures: A case study," in *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, vol. 6. IEEE, 2010, pp. V6–718.

[5] L. Hakulinen, "Card games for teaching data structures and algorithms," in *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*. ACM, 2011, pp. 120–121.

[6] A. I. Wang and B. Wu, "Using game development to teach software architecture," *International Journal of Computer Games Technology*, vol. 2011, p. 4, 2011.

[7] K. Claypool and M. Claypool, "Teaching software engineering through game design," *ACM SIGCSE Bulletin*, vol. 37, no. 3, pp. 123–127, 2005.

[8] P. Gestwicki and F.-S. Sun, "Teaching design patterns through computer game development," *Journal on Educational Resources in Computing (JERIC)*, vol. 8, no. 1, p. 2, 2008.

[9] N. E. Cagiltay, "Teaching software engineering by means of computer-game development: Challenges and opportunities," *British Journal of Educational Technology*, vol. 38, no. 3, pp. 405–415, 2007.

[10] H. Du and L. Shu, "Teaching computer graphics in digital game specialty," in *Information Computing and Applications*. Springer, 2011, pp. 91–97.

[11] J. Linhoff and A. Settle, "Teaching game programming using XNA," *ACM SIGCSE Bulletin*, vol. 40, no. 3, pp. 250–254, 2008.