Academic game development: practices and design strategies for creating STEM games

Mark Edwin Stenerson
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/etd
Part of the Cell Biology Commons, Computer Sciences Commons, and the Instructional Media Design Commons

Recommended Citation
Stenerson, Mark Edwin, "Academic game development: practices and design strategies for creating STEM games" (2012). Graduate Theses and Dissertations. 12556.
http://lib.dr.iastate.edu/etd/12556

This Thesis is brought to you for free and open access by the Graduate College at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Academic game development: practices and design strategies for creating STEM games

by

Mark Edwin Stenerson

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Human Computer Interaction

Program of Study Committee:
Eve Syrkin Wurtele, Major Professor
Steven Herrnstadt
Anne Thompson

Iowa State University
Ames, Iowa
2012

Copyright © Mark Edwin Stenerson, 2012. All rights reserved.
DEDICATION

I would like to dedicate this thesis to my mother, Jean. Without her constant support and encouragement, I would not have been able to complete this work. Love you, Mom!

I would like to thank my friends and family for their support during the writing of this work. In particular, I would like to thank my brother, Matthew, for taking the time to help me edit.
# TABLE OF CONTENTS

**LIST OF FIGURES** ................................................................. v

**ACKNOWLEDGEMENTS** ............................................................. vii

**ABSTRACT** ........................................................................... viii

**CHAPTER 1. INTRODUCTION** ..................................................... 1

**CHAPTER 2. REVIEW OF LITERATURE** ....................................... 3

2.1 Games In The Classroom ......................................................... 3

2.2 Programming Educational Paradigms ........................................... 5

2.3 Using Software to Teach Programming ....................................... 7

2.3.1 Alice ............................................................................ 7

2.3.2 Scratch ........................................................................ 10

2.3.3 Kodu ........................................................................... 12

2.4 Using Games to Teach Programming ......................................... 14

2.4.1 Code Hero ..................................................................... 14

2.4.2 Light-Bot ...................................................................... 15

**CHAPTER 3. DEVELOPING A VIDEO GAME IN ACADEMIA** .......... 18

3.1 Choosing a Game Engine ........................................................... 19

3.2 Creating the User Interface for an Academic Game ....................... 22

3.2.1 A Modular System With Meaningful Names .............................. 23

3.2.2 Concurrency .................................................................. 23

3.3 The BioLog: Bridging the Gap Between Game and Classroom ........ 24

3.4 Conclusions ........................................................................ 28
CHAPTER 4. TEACHING-ASSISTED LEARNING USING SANDBOX

4.1 Consequence-Free Environment .................................................. 29
4.2 Objects-First Approach .............................................................. 30
4.3 Designing the Level ................................................................. 31
  4.3.1 Conditional Statements ....................................................... 31
  4.3.2 Variables ............................................................................. 31
  4.3.3 Looping .............................................................................. 32
4.4 Violence-Free Experience ............................................................. 33

CHAPTER 5. CONCLUSIONS AND FUTURE DIRECTIONS ................. 39

BIBLIOGRAPHY .............................................................................. 40
**LIST OF FIGURES**

| Figure 2.1 | Development environment from Alice by Carnegie Mellon. | 8 |
| Figure 2.2 | Code for “IceSkater.spin” function. | 9 |
| Figure 2.3 | Code editor from Scratch by MIT with a sample script making the character walk back and forth | 11 |
| Figure 2.4 | A screenshot of the code editor in Kodu | 12 |
| Figure 2.5 | A 3D environment in Kodu | 13 |
| Figure 2.6 | A screenshot from CodeHero by Primer | 15 |
| Figure 2.7 | Light-Bot | 17 |
| Figure 3.1 | A snapshot of *Meta!Blast* at the cell level. Players can shrink down to smaller sizes to see more complex processes. | 19 |
| Figure 3.2 | Thought-provoking questions in Meta!Blast | 20 |
| Figure 3.3 | The Web module in *Meta!Blast* is designed to handle any number of simultaneous requests. Design considerations such as this allows for more expandability in the future. | 25 |
| Figure 3.4 | How is in-game educational content, like that seen here in the BioLog in *Meta!Blast*, kept up–to–date? | 27 |
| Figure 3.5 | The BioLog editor allows educators to modify the contents of the in-game BioLog. | 28 |
| Figure 4.1 | The player can only proceed if they can connect the code to the operation of certain objects in the environment. | 30 |
| Figure 4.2 | Visualization leads to level design | 34 |
| Figure 4.3 | In-game inventory design | 35 |
ACKNOWLEDGEMENTS

This work was funded in part by the National Institutes of Health, Science Education Partnership Award 1R25RR025147 for support of Meta!Blast game development. I am grateful for the outreach components of National Science Foundation awards DBI-0520267, EEC-0813570, BIO-IIS0612240, and MCB-0951170 for their support of interdisciplinary education for undergraduate students in computer science, design, and biology, and for educational content. We are grateful for the encouragement and support from Iowa State University including the Department of Genetics, Development of Biology, the College of Liberal Arts and Science, the Virtual Reality Applications Center, and the Plant Sciences Institute.
ABSTRACT

In the last decade, there has been an increase in legislation and funding that aims to get students in the United States interested in fields involving science, technology, engineering, and mathematics (STEM) in order to keep the United States competitive with other countries in these important academic areas. However, making these topics interesting and engaging to students has been challenging. A 2010 report by the Presidential Council of Advisors on Science and Technology concluded that schools in the United States lack passionate teachers and adequate tools to teach these subjects. Due to these deficiencies, “too many American students conclude early in their education that STEM subjects are boring, too difficult, or unwelcoming, leaving them ill-prepared to meet the challenges that will face their generation, their country, and the world” (President’s Council of Advisors on Science and Technology, 2010). Therefore, it could be surmised that something further needs to be done in order to promote these fields to students at a young age.

The Meta!Blast project was developed to provide a medium that lends itself to the comprehension of cell and metabolic biology by placing the student into a virtual plant cell and allowing them to experience plant biology first-hand (Wurtele, 2011). By taking advantage of existing agile development methodologies, Meta!Blast has been designed to meet many of the challenges of developing video games in an academic environment. Using a special editor, educators and researchers can also modify in-game content in an effort to tailor the game to their specific curriculum needs.

Due to the massive, explorative environment in which the game places players, Meta!Blast provides an ideal environment for a variety of other STEM-related mini-games. By leveraging existing methods of current software used to teach computer science, the initial development stage of a mini-game within Meta!Blast called TALUS (Technology Assisted Learning Using Sandbox) has been designed to let players experience different computer programming con-
cepts. The first iteration has shown that an environment can be created that allows players to interact with actual computer code in a fail-safe and non-violent manner; furthermore, it has the potential to augment a player’s existing knowledge of computer programming.
CHAPTER 1. INTRODUCTION

In the last decade, there has been an increase in legislation and funding that aims to get students in the United States interested in fields involving science, technology, engineering, and mathematics (STEM) in order to keep the United States competitive with other countries in these important academic areas. However, making these topics interesting and engaging to students has been a challenging. A 2010 report by the Presidential Council of Advisors on Science and Technology concluded that schools in the United States lack passionate teachers and adequate tools to teach these subjects. Due to these deficiencies, “too many American students conclude early in their education that STEM subjects are boring, too difficult, or unwelcoming, leaving them ill-prepared to meet the challenges that will face their generation, their country, and the world” (President’s Council of Advisors on Science and Technology, 2010). Therefore, it could be surmised that something further needs to be done in order to promote these fields to students at a young age.

One of the emerging theories for spurring student interest in STEM fields involves the play of video games to not only engage students in situations where they need to think critically, but also to inspire students to formulate their own hypotheses and conduct experiments within the subject-matter. In many of the best video games currently on the market (World of Warcraft, Portal 2, Final Fantasy XIII), players find themselves in situations where they need to think critically about their actions and understand the effects that those actions will have on the large-scale system to which their digital avatar belongs. Games have also demonstrated their ability to teach higher-order thinking skills including strategic thinking, interpretive analysis, problem solving, plan formulation and execution, and adaptation to rapid change, all of which are skills that not only U.S. employers seek, but also are the skills needed to compete with lower cost workers in other nations (Federation of American Scientists, 2005). The digital world of video
games offers students the opportunity to experience and experiment in an environment that might not be normally available to them. Examples of these environments can vary drastically from advanced science labs to the surface of the sun. Video games also have the added benefit of having almost no real world repercussions for a student’s mistakes. For example, testing the effects of using water (H2O) in an experiment versus an isotope variant, deuterium oxide (D2O), can deter many high schools because of the costs of deuterium oxide. Student spills and other academic mistakes can make these costs even higher. Comparatively, in a digital lab, a student is free to experiment with different quantities and ratios of a variety of substances without the need to worry about spills, mistakes, and, in some cases, physical danger to themselves or others.

With these factors in mind, the Meta!Blast project was developed to provide a medium that lends itself to the comprehension of cell and metabolic biology by placing the student into a virtual plant cell and allowing them to experience plant biology first-hand (Wurtele, 2011). However, it has developed into more than just a game to spark interest in biology. Due to the massive, explorative environment in which the game places players, Meta!Blast provides an ideal environment for a variety of other STEM-related mini-games. Therefore, this thesis will cover both the development and creation of the Meta!Blast environment and system as well as the initial development stage of a mini-game within Meta!Blast called TALUS (Technology Assisted Learning Using Sandbox) that has been designed to let players experience different computer programming concepts.
CHAPTER 2. REVIEW OF LITERATURE

2.1 Games In The Classroom

If the topic of using games to engage students in STEM fields is to be addressed, one must first ask: can games actually be used to learn?

Kurt Squire addresses a similar question in the first chapter of his book *Video Games and Learning: Teaching and Participatory Culture in the Digital Age* (Squire, 2011). In the chapter, Squire presents a few responses to questions such as, “Why study video games? Aren’t they a waste of time?” In the first response, he concludes that people learn academic content like names of people, places, and terminology through games regardless of whether or not the game is designed for educational purposes. Interestingly enough, when reflecting on some of the more popular video games like World of Warcraft, Pokemon, and Final Fantasy VII, one notices that the true fanatics of these franchises retain an in-depth knowledge about their digital worlds. If a player can absorb this type of information based on a fictitious storyline, one could argue that more meaningful educational content could be delivered to players and absorbed in the same manner. In Squire’s second response, he postulates that games deeply engage those who play them and implement enticing design principles, like overlapping goals and the orchestration of time in completing those goals. In other words, the games that engage players the most are games in which the player is given the ability to manage their goals and complete them in a timely fashion. Squire breaks goals down into short-term goals (60 to 90 seconds), medium-range goals (45 to 60 minutes), and long-term goals (3 or more hours). He goes on to stress the importance of the overlap of these types of goals in convincing people to play the game, making the claim that the short-term goals seduce the player – presumably with small feelings of achievement – until they unknowingly complete a medium-range goal. Squire claims that
using these principles in the instruction of a classroom – even without the use of games – could increase a student’s level of engagement. One possible example of this includes breaking down large, seemingly impossible tasks into smaller sub-tasks and allowing students to budget their own time as well as choose the order in which they complete those tasks. Finally, Squire concludes that video games create participatory situations for players to learn and hone their skills. This is the exact atmosphere that any educator would want from a classroom.

In a 2003 article, James Paul Gee also attempts to answer these questions. He makes the claim that the best game developers incorporate one or more learning principles in order to cognitively motivate people to play the game. Gee feels that these principles are either currently present in schools and could be enhanced with games, or that these principles are sorely needed in schools and that appropriately-designed games could provide them. For starters, good games give information “on demand” and, unlike in schools, don’t give information out of context (Gee, 2003). Plenty of studies in cognitive psychology suggest people are not able to remember or understand information that is presented to them out of context or before they can make effective use of it (Brown et al., 1989; Barsalou, 1999; Glenberg and Robertson, 1999). A well-designed game, especially in the tutorial stages, will offer the player the exact information needed to perform a new task or ability and, in some cases, even remind the player if they appear to be struggling. This ensures that the player learns the fundamental concepts of the game in order to progress to more advanced stages of play. Gee insists this is essential by citing work done by Elman in 1993 which confirms that learners need to be presented with problems in an appropriate order so that initial challenges can establish solid generalizations for subsequent, more complex challenges. However, one of the key statements in Gee’s paper addresses players’ motivation head-on. Gee states that “when motivation dies, learning dies and playing stops.” By citing more cognitive psychology research – this time by Clark in 2003 – Gee supports his belief that games place players in a world where they can engage action from a distance; and this degree of separation actually causes players to feel as if they have stretched into a new space (Clark, 2003). Gee argues that this level of investment seems central to the motivation of a player to stick with and master a game – an element a classroom cannot appropriately create. Finally, Gee stresses that games also motivate players by creating challenges that are
just outside of the players’ competence level. This makes the game difficult, but feasible, as well as rewarding. He contrasts this atmosphere against what currently exists in most schools, where often the lowest common denominator limits the pace for the rest of the class.

The authors of Wurtele et al. (2010) address this question directly when discussing why they chose the serious game format for their cellular biology game, Meta!Blast. They propose that the serious game format offers several key advantages, including the ability to motivate students as well as the ability to provide a safe, explorative environment via a medium familiar to students (Wurtele et al., 2010). Furthermore, educational games provide an environment that can more accurately illustrate many changes in space, time, and size, which is ideal for demonstrating concepts in many scientific fields, including biology. They also suggest that the concept of challenges that scale based on a player’s performance in a game creates an ideal atmosphere for learning not only because of the increase in player motivation but also because of the ability to carefully structure the progression of learning. Most importantly, Wurtele et al. (2010) distinguishes between viewing a simulation of a process and being able to interact with that process first-hand. Simulations, the authors claim, are more of a passive medium that only give players the ability to change a variable so as to view an outcome. However, with dynamic gameplay, storylines, and interactions, video games exhibit the potential of creating an engaging, educational experience with which students actually want to interact.

2.2 Programming Educational Paradigms

In the world of information technology, it becomes increasingly important to not only teach computer programming, but to also teach it effectively. Vujoevi-Janii and Toi (2008) provides a concise breakdown of introductory computer science education including the many programming paradigms that have been created as well as the programming languages that seem to promote each individual paradigm. Although they point out several different paradigms, the most important and widely-used programming education paradigms today are: imperative, object-oriented, functional, and logic. The first programming paradigm – the imperative paradigm – operates on the Von Neumann computer architecture from the 1940s which involves a sequential CPU with separate memory and with data piped in between. While the architectural concept
might be old, it is important to note that modern-day CPUs still use some of these principles. The imperative paradigm focuses on teaching states, sequential orders, and assignments, and is popular in earlier programming languages such as FORTRAN and Cobol. The authors argue that the imperative paradigm, while being the easiest paradigm for novice programmers to understand and one the most popular teaching paradigms, can cause problems if the concepts of assignment and sequence are not thoroughly understood by students. The second programming paradigm is the object-oriented paradigm and is based on simulation. The main idea centers around how the simulation should reflect the real-world environment being simulated. In the real world, objects interact with one another to produce a result. The object-oriented paradigm calls for programmers to define objects and classes that pass information back and forth in order to solve a problem or achieve the desired outcome. The paradigm focuses on teaching abstraction, inheritance, and polymorphism; and it is supported by languages like Java and C++. One could argue that the major benefit of the object-oriented paradigm correlates to its major hindrance. It has been said that switching from a procedural paradigm to an object-oriented paradigm later in a student’s programming education is more difficult than if the student had started in object-orientation (Guzdial, 1995; DeClue, 1996). However, there is also a worry that students will lose the basic ability to make low-level implementations of their classes if they become too focused on object-orientation (Duke et al., 2000). The functional paradigm, based on lambda-calculus, encourages learners to think about the nature of the problem instead of the nature of the underlying computational concepts. However, the functional paradigm is starting to disappear due to the ease of integrating this type of thinking into the imperative and object-oriented paradigms. Lastly, the logic paradigm, based on first-order predicate calculus, is characterized by defining rules of inference and axioms to discover new information about problems as well as how to solve them. However, since both of the paradigms function on concepts in calculus that are beyond most novice-level programmers, both the functional and logical paradigms have started to decline in popularity in introductory programming courses.
2.3 Using Software to Teach Programming

Over the last few years, many different software packages have been created in an effort to help teach and generate interest in computer programming. Before a new solution can be created, one must first look at what has already been done in order to leverage existing techniques.

2.3.1 Alice

One of the major applications for teaching computer programming appears in the freeware application *Alice* by Carnegie Mellon University (Cooper et al., 2000). *Alice* attempts to make programming easier by providing several tutorials as well as a click-and-drag interface for running a series of commands (Figure 2.1). Students can build their own scenes and learn how to program by creating their own stories, small games, and movies. Rather than write actual code, students are allowed to create statements that are more readable to a layperson. The program itself manages the parameters of these statements via a “smart editor” which creates an ideal environment for allowing the student to create syntactically correct, error-free code. This guarantees that the system remains stable while giving the student freedom to learn and experiment.

The most difficult part of *Alice* is understanding how to create animations. To understand this in more detail, one can look at the “IceSkater” character model that comes packaged with *Alice*. The “IceSkater” is, as the name suggests, a 3-dimensional model of an ice skater that the programmers of *Alice* have already animated for users. For example, the function “IceSkater.spin” makes the “IceSkater” model spin. The code for the spin function can be seen in Figure 2.2. The sequence of events is easy enough to understand; however, it is easy to see that writing this function would require quite a bit of thought and an in-depth knowledge of how to create more of a natural movement. This is not something that an introductory-level programmer could achieve.

Although there is a small learning curve when using *Alice*, the program has been useful in teaching introductory computer programming by using the “objects-first” approach. In Cooper
et al. (2003), the creators of *Alice* describe the advantages and disadvantages of using an “object-first” approach rather than the more popular “imperative-first” or “functional-first” methods. In two separate introductory computer science courses, the authors observed a wide range of benefits including a stronger sense of design, a contextualization for objects, a better understanding of encapsulation and inheritance, and an understanding of Boolean types. The one weakness that came up involved student knowledge of syntax. Because of the “smart editor” used in *Alice*, students never experienced the many errors that arise due to missed parentheses, brackets, semicolons, etc. To dispel this shortcoming, the authors mention that students generally do not have a problem mastering syntax after they have transitioned from *Alice* to an actual programming language like Java or C++. Also, the authors conducted a small study in which eleven weak computer science students (defined as students who were not ready for calculus and had no prior programming experience) were taught introductory programming using *Alice* while a control group from the same population was taught using traditional methods. The results of the study showed that 91% of the *Alice* students opted to
Figure 2.2  Code for “IceSkater.spin” function.
continue to the next level of programming courses the following semester versus only 10% in the control group. While the validity of this study might be open to sceptic interpretation, it does warrant further research with Alice and similar systems.

2.3.2 Scratch

Another freeware application that has caught the eye of school systems is Scratch (MIT Media Lab, 2011). Scratch was developed by the Massachusetts Institute of Technology Media Lab for the purpose of teaching introductory programming concepts and, like Alice, allows students to create their own games and animations. A noticeable difference when first comparing Scratch to Alice is the interface that Scratch uses to create programs. Scratch categorizes different actions using a variety of colors, employing a puzzle-like scheme to fit those actions together (Figure 2.3). In Resnick et al. (2009), the creators of Scratch explain that their design scheme was centered around how kids play with Legos. Students are allowed to “tinker” around with different blocks in order to build scenes and games. Like Alice, the code editor for Scratch could be considered a “smart editor” in the sense that students are not allowed to create code that crashes the system since each of the pieces fit together in a very specific way. Unlike Alice, though, the graphical side of Scratch is far easier to understand. Scratch is designed to utilize two-dimensional sprites rather than the more complex three-dimensional models characteristic of Alice. This makes Scratch easier to understand by a novice and may explain why Scratch has created a massive online following of users, now able and proud to publish their own works of art. The creators of Scratch make the claim that, as of 2009, more than 1,500 projects are uploaded to the site each day and, while adults also publish, the core audience on the site seems to fall between eight and sixteen years old, peaking around twelve (Resnick et al., 2009). Even more impressive is the creation of online Scratch companies for collaboration and development of Scratch games. These “companies” are run by Scratch users – some reportedly as young as eleven years old. In the paper, the creators of Scratch use the example of a company created between users from England, New Jersey, and Ireland to show that the level of collaboration with Scratch extends across the globe.
Figure 2.3  Code editor from Scratch by MIT with a sample script making the character walk back and forth
2.3.3 Kodu

From the private sector comes Kodu, a program that implements a visual programming language developed by Microsoft specifically for teaching programming (MacLaurin, 2011). Unlike Scratch and Alice, Kodu seems to have an easier-to-learn user interface (Figure 2.4), more compelling graphics (Figure 2.5), and an almost non-existent learning curve for creating three-dimensional games.

![A screenshot of the code editor in Kodu](image)

**Figure 2.4** A screenshot of the code editor in Kodu

In Stolee and Fristoe (2011), a study was conducted on 346 sample Kodu programs in order to answer three research questions: 1) what computer science concepts can be taught or expressed in Kodu?; 2) how much time can users spend programming their games in Kodu?; and 3) how often does each computer science concept appear in these games?

In terms of teaching, the authors postulate that Kodu has the potential to teach a wide variety of computer science concepts including variables, scope, boolean logic, objects, and control flow. While some actions repeat by default, one noticeable downfall to Kodu is the lack of support for defining basic control structures like loops and functions. Since it fails to address
these introductory concepts, one could argue that Kodu skips fundamental instruction which could create problems in a student’s base understanding of how a program works.

The study did, however, yield some positive results. While the authors admit to the need of testing to confirm that learning took place, many of the programs that they sampled were large and complex worlds that exhibited a variety of programming concepts. The study showed that almost three-fourths of the games that were studied contained programmed characters or, in other words, characters that had code created to make them more interactive than their original packaged state (with an average number of rules per character of 5.14). Therefore, one could conclude that users not only create these characters, but also take the time to make the characters more complex. The most interesting result of this study centered around the time spent in the Kodu environment itself. The study showed that, on average, users spent around 25 minutes at a time using Kodu. During that time, users tended to edit code an average of 12.7 times and played their game an average of 13.7 times. The authors believe this indicates that users rapidly switch back and forth from editing to playing in an effort to test their new code as they develop. This serves as a common technique for students learning how
to program (Hanks and Brandt, 2009).

2.4 Using Games to Teach Programming

While these are all unique examples of software platforms for entry-level programmers to create games, these applications lack any actual gameplay and, therefore, cannot be classified as video games. However, there are some available games in the process of addressing this issue.

2.4.1 Code Hero

One such game is called Code Hero and is currently in development by Primer Labs in San Francisco, California (Peake, 2011). Written with the Unity game engine, Code Hero, a first-person shooter, attempts to teach players how to program by letting them launch pre-written snippets of code at objects in the environment with the “Code Gun” in order to observe how objects will react and, eventually, to solve puzzles. Players are even allowed to write their own code so that they can customize their interactions (Figure 2.6).

One of the negative criticisms of Code Hero arises with the fact that the players, presumably students in an educational setting, would have to shoot objects in order to play. With more and more research indicating a causal link between violence in games and increased aggression (Anderson et al., 2010), the shooting mechanic of Code Hero used in the classroom is a cause for concern. Furthermore, the ability for students to write their own code in-game creates the possibility for the students to crash the game when they try to run code that contains an error. In essence, this feature inhibits one of the main benefits of learning in a video game: the ability to experiment in a consequence-free environment.

One interesting aspect of Code Hero is that, unlike Alice, Scratch, and Kodu, the game requires students to write actual game code in order to play through the environment. This eliminates the concern that students will lack syntax comprehension later in their education and opens the door for the developers of Code Hero to literally teach whatever they want via the game. This does, however, raise a concern about level of difficulty within the game. The creators of Code Hero claim that novice-level programmers can play their game, implying
that the tutorial section will guide them through initial difficulties. Until the game can be thoroughly analysed, one could argue that the game does not address the difficulties of initially learning how to think like a programmer. While the aforementioned programs have their own drawbacks, they all attempt to address the concern of generating student interest by letting them “test the water” rather than push the student straight in, as Code Hero seems to do.

2.4.2 Light-Bot

The Flash game Light-Bot, hosted by Armor Games, is a fun game that uses programming-style logic to pilot a small robot around a puzzle grid (Yaroslavski and McNeely, 2011). Players give the robot a list of commands and, after the player hits the “run” button, the robot executes the commands one-by-one (Figure 2.7(a)). If all of the lights are lit when the robot finishes his tasks, the puzzle is solved. If not, the player tries again. This game refrains from punishing the player for experimenting and does not involve anything that may model violence. One could also argue that it teaches students the beginnings of coordinate systems. In the latest version,
dubbed *Light-Bot 2*, a tutorial system shows the player how to pilot the *Light-Bot*, run code recursively, and use conditions to manage what the Light-Bot does or does not do. Also, the speed at which the commands are executed can be slowed down while highlighted tiles provide a visualization to the step-by-step process that the *Light-Bot* performs (Figure 2.7(b)). However, the creators of *Light-Bot* have yet to create a game mechanic that promotes concepts like code syntax, inheritance, variables, etc. While the game is enjoyable, it would take a fair amount of work to warrant regular classroom use in an introductory programming course.
(a) A screenshot of a puzzle

(b) A screenshot of Light-Bot moving through the puzzle

Figure 2.7  Light-Bot
CHAPTER 3. DEVELOPING A VIDEO GAME IN ACADEMIA

The emerging field of educational gaming (also known as serious gaming) shows tremendous promise in addressing the lack of student interest in many of the fundamental STEM fields. It is believed by many scholars that, under the right circumstances, games not only have the ability to entertain students, but also to educate them. This notion, along with the National Science Foundation, the National Institute of Health, and other sources of financial support, has spawned the creation of a variety of game projects in academic settings whose goals are to create fun and interactive games to use as teaching tools in the classroom. However, unlike mainstream video games produced by large companies with expansive budgets, academic game development teams are constantly changing as graduate and undergraduate students that normally comprise them graduate and leave to pursue their careers. Academic teams also encounter newer research regularly that might force them to rethink their game design in the middle of development, costing precious time and money.

Meta!Blast is a real-time 3D action-adventure game designed for high school and college level students that puts a player inside a virtual plant cell (Wurtele, 2011). By immersing players into such an environment, the developers of Meta!Blast anticipate that players will come to a greater understanding of the cell than they could from traditional diagrams and textbooks (Figure 3.1). The current version of Meta!Blast allows players to travel around the cell in their “bioship” and answer an assortment of thought-provoking questions stored in data capsules scattered throughout the cell (Figure 3.2(a)). We detail some of the recent innovations to Meta!Blast, including how we have begun designing Meta!Blast for easy updating by educators and researchers, and explain how the development team has designed the game to meet some of the challenges that arise when developing a game in an academic environment.
3.1 Choosing a Game Engine

When starting any game-related project, one of the first substantial decisions revolves around which game engine to use. From 2007 to 2012, several impressive game engines have been made available to the public by many different companies, like the *Unreal* game engine (Epic Games) and the *Unity3D* engine (Unity Technologies). However, these engines all vary in several key areas such as technical capabilities, the target platforms for game publishing, and, of course, licensing costs. In order to choose an appropriate engine for an educational game like *MetaBlast*, all of these factors become important. We exemplify these considerations with comparisons of the *Unreal* and *Unity3D* game engines.

First, it is crucial to look at the length of the graphics pipeline for a particular engine. In other words, how long does it take and how difficult is it to create a 3D object model, bring it into the engine, and get it to a point where it becomes usable in game development? For any game – especially one being developed by an academic institution – it is important to be able to rapidly prototype and test ideas in the game environment without wasting precious time.
(a) An example of an in-game question.

(b) Teachers can choose to include feedback for each question in the game, allowing students to reflect on their answer.

Figure 3.2 Thought-provoking questions in Meta!Blast
time and resources. Since many of the art assets for Meta!Blast have already been developed, another consideration arises with the engines ability to accept existing 3D models without the need for excessive and complicated conversion. In order to effectively test this, the 3D model of the Bioship – the main player vehicle of Meta!Blast (Wurtele, 2011) – was exported from the 3D modeling program Maya, which is the primary modeling system used by Meta!Blast artists.

The Unreal engine was able to import the model and textures in a matter of minutes. However, when the models were imported, the individual pieces of the model split into separate files in the engine. Models would either need to be manual reassembly in the Unreal engines world editor (which is where components of the 3D environment are pieced together), or assembled with an external script before they could function as a single unit. Therefore, the more complex the model, the longer the graphics pipeline will be with the Unreal engine.

The Unity3D engine was able to import the model and texture at a comparable rate to the Unreal engine. However, unlike Unreal, Unity3D has the ability to open up the raw data files that artists use to save their work. This unique ability allows artists to update shared assets and have Unity3D automatically update the attached model and textures. Also, Unity3D imports each Maya file using the exact coordinates and placement that Maya uses. This means that each model imported from Maya0 assembles in the exact the way the artist designed it allowing for entire levels to be built by the artists with little-to-no need for adjustment inside the game engine. This makes the Unity3D graphics pipeline almost automatic.

When choosing a game engine, one must consider the target platforms for the game. For instance, will the game be played on a Windows PC or a Mac? Or is an iPad game best? Both Unreal and Unity3D development environments support creating games for Windows, Mac, iPad, Android, Playstation3, and XBox. With this ability, developers can create games that reach a much wider audience.

None of this information matters without the right price. Business projects and academic development projects must consider their budgets for software. As of 20112, Unity3D has two different licensing options: the free “Indie” license and the Pro license ($650/seat) which grants developers more abilities, including level-of-detail support, video playback, real-time shadows, and more (Unity Technologies, 2012). Meanwhile, the fully featured development kit for the
Unreal engine is available, as of 2012, to educational institutions for free as long as it is for non-commercial use (Epic Games, 2012). This aspect can make the Unreal engine very appealing to low-budget projects in academia.

3.2 Creating the User Interface for an Academic Game

A general rule of thumb in business states that only one constant exists: change. The same can be said for the “business” of academic software development. Several software development businesses experiences cases where additional content and updates need to be added to a program after the release of the initial version of the program. Martin (2008) highlights many key aspects of computer programming that aid in making software that can be easily changed and adapted based on the changing needs of clients and developers. He argues that software development should be “cleaned up” in many ways, ranging from naming objects meaningfully to design of the entire system as a whole. With this, the code is written so that it can not only be easily updated but also be easily enhanced and updated by programmers other than the original developer.

These principles are just as important, if not more important, in an academic setting. On a large project, undergraduate and graduate student developers come and go throughout the development process, making it essential that any game built in an academic setting be programmed in such a way that makes it easy to hand the project off to a new generation of developers and makes it feasible to collaborate remotely with other developers. One reason software is not cleaning and efficiently written is lack of training in good programming practices. Furthermore, as Pollice (2006) hypothesises, one of the main reasons why “dirty code” is created directs from pressure on programmers due to a lack of time. Therefore, the programming structure of Meta!Blast was designed to create a system that would be transparent to a new developer on the project. One example is the user interface code – the code that not only visually creates menus for the player but also operates a menu based off of a player’s interactions with the game. The following breakdown illustrates how the concepts of Modular Systems, Meaningful Names, and Concurrency were effectively used to create a stable and extensible system.
3.2.1 A Modular System With Meaningful Names

Two of the principles that Martin (2008) highlights includes modular systems and meaningful names. He states that “an optimal system architecture consists of modularized domains of concern.” Simply, the code should be split into separate sections, each with a specific purpose. Those sections, or modules, interact with each other in order to complete a certain task and, in essence, produce the finished product.

A modular system architecture is meaningless without objects with meaningful names. It is important and helpful to new programmers on a team to be able to easily read existing code. This means that variables and functions are named so that their purpose can be ascertained at first glance. When creating a game, the same principle also applies to the objects in a scene or level. For instance, having an object named “Object X” conveys little information. Instead, if the object was named “Cow”, the programmer instantly knows more about it.

When designing the scenes in Meta!Blast, objects were appropriately named for the benefit of new developers. The user interface code was split into several different modules, each named specifically for the functions that they perform. For instance, the module that manages the main menu screen – the first screen the player sees in the game – is aptly named “MainMenu”. This way, if a change or an update to this screen is needed, a new programmer on the team immediately knows where to look to start implementing the change.

3.2.2 Concurrency

A third principle is concurrency (Martin, 2008). From a traditional computer science standpoint, concurrency usually relates to multiple threads or processes that all run simultaneously, potentially interacting with each other or accessing the same dataset. In game design, concurrency can also refer to multiple game objects that all simultaneously interact with each other and their environment in a given scene or level. The challenge in programming concurrency lies in making sure that multiple threads can access the same resources simultaneously without corrupting any data or interfering with each other. For instance, in Meta!Blast, the Web module is designed to manage all interactions that the game has with the server. During gameplay, many
different objects access the Web module in order to send and/or receive information from the
Meta!Blast server. However, in Unity3D an object in the scene must wait for the response from
the server before continuing with its current task. This proves problematic if multiple objects
need information from the server at the same time. The Web module was therefore designed
using a callback system that allows an object to specify what to do with data returned from the
server. Figure 3.3 illustrates this callback system using the Questions module, which manages
the in-game questions, and the Stats module, which is responsible for reporting user statistics
like scores and demographics. In this example, the Questions module needs to retrieve a set
of new questions from the server. At the same time, the Stats module is attempting to upload
the player’s login time. The two module can submit the data to the server via the Web module
and specify that they want the Web module to send the returned data back. In this instance,
the Questions module is using its loadQuestionsCallback function and the Stats module, since
it isn’t downloading any information, doesn’t require a callback. The Web module then creates
temporary objects called “Stream” which will submit the requests to the server and then wait
for and handle the responses, allowing the Web module to remain active in order to receive
more requests from other objects.

3.3 The BioLog: Bridging the Gap Between Game and Classroom

Arguably the biggest challenge that educational game developers face is how to integrate
content into gameplay so that the student is not only engaged in play but also learning the
desired content. This becomes increasingly more challenging when developing games that relate
to rapidly changing scientific fields, as new discoveries and new research are made available.
Many articles on educational gaming state that detailed development is required for games used
in the classroom (Dondlinger, 2007). Since it is unrealistic to expect an educator or researcher
to modify the game code, how can Meta!Blast keep up-to-date with the latest research in fields
like metabolic biology, cell biology, and bioenergy without having to completely change the
game every time a new discovery is made?

Such considerations lead to questions such as, “can gameplay and game content be separated
in such a way that allows educators and researchers to modify game content without having
The Web module in Meta!Blast is designed to handle any number of simultaneous requests. Design considerations such as this allows for more expandability in the future.

Zimmerman and Fortugno (2005) state that “one feature of all good educational games is a marriage of form (gameplay) and content”. In contrast, Gee (2007) concludes that “in video games, unlike in novels and films, content has to be separated by gameplay.” Clearly, delineating the difference between game content and gameplay was crucial in addressing how to enable content change in the context of Meta!Blast.

The term “game play” has many different definitions and interpretations. For instance, Bjork and Holopainen (2004) define game play as “the structures of player interaction with the game system and with other players in the game” whereas Lindley et al. (2008) define game play as “the experience of interacting with a game design in the performance of cognitive tasks, with a variety of emotions arising from or associated with different elements of motivation, task performance and completion.” One unifying theme seems to arise from the definitions of gameplay: interaction. Almost all definitions of “game play” either mention a player’s interactions with the game or allude to game play being tied to what the player experiences in the game. Therefore, a broad view of game play could entail “the interactions that the player has with
the game.”

The dictionary defines content as “something that is to be expressed through some medium, as speech, writing, or any of the various arts.” Thus, content with regards to an educational game can be considered “what the developers of the game are trying to teach to their audience via their game.”

If game play is viewed as the interactions that the player of a video game has with the game and the game content is viewed as “the information that the game developers are trying to convey to players”, then one can postulate that game play and game content should act independent of each other. In fact, Gee (2007) contends that “content in a game sets up, but does not fully determine, game play.” The example that Gee gives involves the controversial game *Grand Theft Auto: San Andreas*. While not viewed as an educational game in the traditional sense, Gee points out that the content of *Grand Theft Auto: San Andreas* involves concepts like poverty and crime while the game play involves problem solving situations like evading cars as you ride a bicycle through town. He concludes that if one were to change the game play by taking pictures of people rather than killing them, the problem-solving aspects and difficulty of the game would remain relatively unchanged.

When reflecting on the definition of game content, it can be clearly inferred that the content of *Meta!Blast* primarily centers around plant cell biology. One consideration in creating the initial release came with how to deliver more complex, vocabulary-rich content than what would be provided by the player simply flying through the cell. One possibility proffered designing *Meta!Blast* to pair with a textbook that players could reference when they wanted more in-depth information about a specific biological concept. This feature, however, would require creating and/or providing the students with such a textbook. In addition, the “flow” of the game would be disrupted if a student had to continuously reference a textbook while playing. Flow, as defined by Csikszentmihalyi (1991), is “the state in which people are so involved in an activity that nothing else seems to matter.” It goes without saying that, in the context of an educational game, players may benefit more when involved in the interactivity of the game without the reminder that they are, in fact, learning. As such, the BioLog was created (Figure 3.4) – a virtual in-game database that would allow students to click on objects in the
Figure 3.4  How is in-game educational content, like that seen here in the BioLog in *Meta!Blast*, kept up-to-date?

cell and find more detailed information about their environment without taking their focus off of the game.

During the initial design phase of the BioLog, it was determined that requiring players be dependent on a working internet connection was unrealistic. Therefore, the contents of the BioLog are stored on each player’s computer in an XML (eXtensible Markup Language) file and loaded into the game when the game initially loads. This created a unique separation of game content and gameplay. As long as the format and validity of the XML file was maintained, the game could load and display the contents of the file regardless of the information it contained.

Through a prototype, custom editor that comes included with the game (Figure 3.5), educators are given the ability to create, edit, and in some cases, delete BioLog entries, thereby allowing the incorporation of curriculum-specific information into the game without the possibility of compromising the validity of the XML file. This way, educators can tailor their version of the BioLog to the abilities and background of their students.
Figure 3.5 The BioLog editor allows educators to modify the contents of the in-game BioLog.

3.4 Conclusions

In the business of software development, it is not uncommon to see companies with an abundance of resources creating complex and sizeable products. If the world of academia hopes to create software packages of similar size and quality – such as large-scale educational video games – it becomes absolutely essential that rigorous developmental practices be followed. The developers of large educational games like Meta!Blast must account for the likelihood that their work will be continued by others and that, due to expanding research, the game being developed must allow for a certain amount of flexibility in the content of the game. These concepts, coupled with use of a stable, mainstream game development engine, will provide a fantastic foundation for academic developers to start creating the next generation of video games in the classroom.
CHAPTER 4.  TEACHING-ASSISTED LEARNING USING SANDBOX (TALUS)

By leveraging the existing research described in Chapter 2, an introductory level of a video game was created that attempted to teach players about rudimentary computer programming concepts. The game, called TALUS (Teaching-Assisted Learning Using Sandbox), was designed with three major goals in mind: creating an environment that leveraged current research in order to help students visualize new concepts; creating a consequence-free environment in which students can learn and experiment; and finally creating a non-violent game for use in the classroom.

4.1 Consequence-Free Environment

One of the main benefits of learning in a video game consists of the ability for the student to experiment without any real-world consequences (Gee, 2007). When designing TALUS, one of the main challenges was to figure out how to give the player certain control over their environment without the possibility of the game crashing. Therefore, the first task in designing the game was to find out how to implement this kind of control. The Unity3D game engine provided the functionality of running user-generated code using the eval method. However, without the ability to account for certain common programming logic errors (like infinite loops, for instance), this option was not viable as it had a higher likelihood of user error. While software platforms like Alice equip users with total control over their environment – including the ability to write detailed and advanced code – it is impractical for a game environment to contain this quantity of new concepts, all of which must be carefully controlled and incremented in the game design. It was instead decided that player would be presented with code to
which they could make small changes in order to produce a desired result or navigate through a particular challenge. This way, players would not only see the correct syntax, but also make their changes in such a way that would minimize the possibility of system compromise. Figure 4.1 shows an example of a for loop that is encountered at a later stage in the game. In this example, a player has control over the integer variable $X$ in order to control how many iterations are performed; but they cannot otherwise change the structure of the code.

![Figure 4.1](image)

**Figure 4.1** The player can only proceed if they can connect the code to the operation of certain objects in the environment.

### 4.2 Objects-First Approach

Given the gaming environment, the object-oriented paradigm discussed in Chapter 2 was the most appealing and obvious paradigm to use in TALUS. As with objects in the real world, objects in a computer program have a variety of properties and functions that are modified and executed every day by a wide range of other objects. Objects in a video game are no different.
Also, the strengths observed in Cooper et al. (2003) when using an “objects-first” approach in the Alice environment to teach programming clearly outweighed the suspected weakness of students not comprehending syntax. Furthermore, since students would interpret code rather than create it, the potential threat of students not understanding syntax would be minimal.

4.3 Designing the Level

The next step in the design of TALUS was to determine what the first level was going to teach and, specifically, how the game was going to teach it. Research conducted by Dr. Lindsey Ford at the University of Exeter in the United Kingdom proved useful (Ford, 1993). Using a simple animation editor, Dr. Ford found several visual similarities in the ways in which beginner-level programmers visualized the concepts taught in class. Based on her results, it was decided that conditional statements, “while loops”, “for loops”, and the beginnings of variable storage would be present in the first level.

4.3.1 Conditional Statements

When asked how they visualized conditional statements, many of the students in Ford (1993) drew structures that involved multiple branching paths. Therefore, it was decided that a small maze would be created in the game that would attempt to teach players the beginnings of conditional statement logic. Figure 4.2(a) shows an example of conditional statement visualization from Ford (1993) while Figure 4.2(b) shows the recreated 3D scene in TALUS. In order to proceed, the player must interpret the room’s conditional logic and choose the correct path. If they choose incorrectly, they become lost in the fog and must start again at the beginning of the maze.

4.3.2 Variables

In many adventure or role-playing games, players are given an inventory in which they may place different items they find on their journey. Some games, like World of Warcraft, limit how much a player may carry at one time (Figure 4.3(a)). After looking at more of the visualizations in Ford (1993), it was hypothesized that this simple and timeless game mechanic could prove
useful in teaching the sizes of different types of programming variables (Figure 4.3(b)). At certain points in TALUS, players find themselves in a situation where they need to change a variable attached to a particular object so they can proceed. In order to change a variable, the player must store the variable into TALUS’s memory bank, displayed on the left side of the screen (Figure 4.3(c)). At the same time, they may only store the variable if they have enough free memory to do so. In the first level, players must differentiate between boolean variables – which occupy inventory slot – and integer variables – which occupy four slots. These sizes are congruent to boolean and integer variables in C++ which occupy one byte and four bytes respectively. Like the item inventory in World of Warcraft, limiting the amount of memory slots that a player has at any given time forces them to manage what variables they have stored and, in essence, helps them understand the size difference between the types of variables.

4.3.3 Looping

Adding the concepts of “for loops” and “while loops” introduced an interesting challenge. Ford (1993) showed that students visualized loops as objects that were turning or rotating in some way (Figure 4.4(a)). Two different implementations of loops were hypothesized: a physical representation of a loop in the form of an additional chamber, and a singular object that turned in the environment. To create a physical representation of a loop, the “while chamber” was developed (Figure 4.4(b)). Upon executing the main method in this chamber, a player was forced to traverse a looping path, modifying a variable with each iteration. The player eventually progressed once the variable had reached a certain value. Similarly, a “while loop” in any programming language continues to loop until the condition that manages the loop is satisfied. While the game never tells the player about this connection, it is anticipated that this room would be used by a teacher to explain “while loop” execution. At a certain point in the level, a player must turn a giant key in order to open a corresponding door. Each key represents a “for loop”. Using a screen like the one seen in Figure 4.1 and by changing the variables that govern the minimum and maximum values of each loop, a player can affect how many times the loop iterates, directly affecting how much the key physically turns (Figure 4.5(a)). Obviously, if the minimum contains a value greater than the maximum, then the loop will not do anything.
Understanding how these variables affect the execution of the loop is essential to understanding the state of the program after the loop finishes and is a key concept needed for introductory programming students.

4.4 Violence-Free Experience

The primary method of a player’s interaction with the virtual environment was also crucial in the design of this game. Since research suggests that violence in video games can increase aggression in players (Anderson et al., 2010; Swing et al., 2011), it was decided that the game should not involve any weapon or violent action of any kind. The player should also have a fair amount of time when viewing a problem or piece of code to avoid feelings of pressure. The “Scanner” ability was developed to solve this problem. The “Scanner” allows a player to scan certain objects in the game environment in order to view any variables or methods that they might be able to use to get past a particular problem. Figure 4.6(a) shows a player scanning a treasure chest to figure out how to open it. The “Scanner” provides the player with an interaction similar to the interaction that a programmer would have in a class. Once the “Scanner” displays the results (Figure 4.6(b)), the player can see that each object in the environment is capable of having public methods and variables – all with which the player can directly interact – as well as private methods and variables – when the player can view but not directly interact with. The idea behind this design suggests that players begin to understand the computer programming concept of scope in regards to what can be run outside of the object scope (public) versus what the object runs internally (private).
(a) A student visualization of a conditional statement

(b) The player must interpret the conditional logic to proceed

Figure 4.2 Visualization leads to level design
The inventory in TALUS revolves around collecting variables and storing them in TALUS’ memory bank. Each node in memory relates to one byte of storage, allowing students to experience the size differences of certain variable types.

Figure 4.3 In-game inventory design
(a) Student visualization of a “while loop”

(b) Players enter this chamber and are forced to loop down colored hallways until a certain condition is satisfied and the treasure is revealed.

Figure 4.4 Experiencing “while loops” in TALUS
Figure 4.5 Experiencing “for loops” in TALUS
(a) A player scans a treasure chest

(b) After scanning an object, the player can view and manipulate certain properties of that object

Figure 4.6 Scanning objects to manipulate them
CHAPTER 5. CONCLUSIONS AND FUTURE DIRECTIONS

In recent years, the concept of creating video games for use in the classroom has become more mainstream and is slowly starting to garner more acceptance. With the government urging educators to find new ways to get students interested in the STEM fields, more and more educational game development projects are being started in universities all over the country. However, it can be difficult to create large scale, maintainable video games in an academic environment where graduate and undergraduate developers are constantly changing and new discoveries are being made each day.

Meta!Blast, an educational game about plant cells currently in development, addresses many of the problems that arise in developing a video game in academia. By leveraging existing techniques used in business-level programming, the Meta!Blast code has been cleanly written to prepare for the next generation of developers involved with the project. The structure of Meta!Blast allows for educators to customize in-game content via a custom editor in order to make curriculum-specific modifications. It has also been created to serve as an ideal platform for delivering a variety of mini-games, encompassing many different STEM fields.

TALUS (Teaching Assisted Learning Using Sandbox) is one such game. The first level has shown that an environment can be created to allow players to interact with actual computer code in a fail-safe and non-violent manner maintaining the potential to augment a player's existing knowledge of computer programming. While it goes without saying that this is only the first step in creating a safe and beneficial game to use in a classroom, the proper foundation for the game has been established so that iterative development and in-depth, user-centered testing can begin.
BIBLIOGRAPHY


President’s Council of Advisors on Science and Technology (2010). Prepare and inspire: K-12 education in science, technology, engineering, and math (stem) for america’s future. [http://www.whitehouse.gov/sites/default/files/microsites/ostp/pcast-stemed-report.pdf](http://www.whitehouse.gov/sites/default/files/microsites/ostp/pcast-stemed-report.pdf).


