DESIGNING AND CREATING A GAME ENGINE FOR USE IN THE CLASSROOM

Robert Spears, Cary Rivet, Stephen Killingsworth, Ashok Kumar and Jim Etheredge

School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, Louisiana, USA

ABSTRACT

This paper describes the creation of a 2D game engine, the ULL Introductory Game Engine (UIGE), for use in an introductory video game programming course as part of an undergraduate computer science curriculum. Having the right tools available can make illustrating the concepts of game development and design substantially easier. In creating the engine, a number of factors such as ease of use and accessibility, were considered. Furthermore, course instructors must determine how much assistance (in the form of tools for the engine) is too much assistance, as students may miss core principles if development with the engine is made too abstract. Successful implementation of tools like UIGE allow students to produce games quickly and the concepts of game development to be taught more effectively.

KEYWORDS

Game Design, Game Engine, Game Class, XNA, C#, DigitalRune

1. INTRODUCTION

Teaching game design, programming, and theory is a challenging task. Illustrating the concepts useful in game programming is a multistep process that involves much more than simply illustrating how to use code. Other topics such as game management, asset and pipeline management, and group dynamics must also be incorporated. The concept of adhering to deadlines and milestones is also a crucial part of the process, as most of the projects are iterative min nature. Missing one milestone immediately impacts the schedule of the rest of the project. Therefore, the goal for course administrators is to create an environment where coding and designing are as easy as possible. This is crucial, as the time participants have to complete their video game projects is dramatically smaller than what is allotted to game developers in the industry.

One way to achieve high productivity in a video game project course is to provide its participants with a game engine that is both accessible and equipped with all the necessary tools needed for development. The engine can be relatively flexible and modifiable, but should guide its users to a particular game type. This gives the developers of the engine the opportunity to establish a solid foundation upon which its users can expand. The inclusion of useful classes, functions, and overall structure allow for a greater efficiency in the development process.

This paper discusses the planning and implementation of such an engine, dubbed the ULL Introductory Game Engine (UIGE) [wee-gee], for use by computer science majors in an introductory game development course. Section 2 of the paper provides an overview of the engine's characteristics, and explains how it was tailored toward students of a particular skill level. Section 2 also outlines the general layout of the course in which the engine was used. In Section 3, UIGE is described with greater depth, with its creation presented in a chronological fashion. Section 4 exhibits sample projects created with the engine, and offers a timeline of the projects' development. In Section 5, works related to UIGE's construction are visited. Finally,Section 6 gives some conclusions on the production and use of UIGE, and highlights how continued development of tools like UIGE will help improve future versions of the course. This paper assumes that the reader is familiar with the object oriented paradigm, as well as general video game and programming concepts and technologies

2. PROJECT OVERVIEW

The School of Computing and Informatics at the University of Louisiana at Lafayette offers its Computer Science students a number of concentrations to choose from. The video game concentration is one such option. It offers those enrolled a number of game design focused electives such as Artificial Intelligence, Graphics, Game Algorithms, as well as 2D and 3D project courses. These courses are offered in addition to the core computer science curriculum. UIGE was designed and implemented specifically to be used by students taking the course. Also known as An Introduction to Video Game Design and Development, the course is the first those in the video game concentration are required to take. Participants in the course have already completed a number of courses in the main curriculum, and as such, they possess extensive knowledge on advanced data structures, efficient searching and sorting algorithms, and memory management techniques. However, most students enrolled in the course have not yet been exposed to the paradigms of video game development and design.

2.1. Course Layout

During the Fall semester of 2012, students in the class were assigned two video game projects. The first was a solo effort using YoYo Game's GameMaker: Studio [1], while the second was a group project using UIGE. The first project ran for three weeks, and its objective was to give students a brief introduction to game design theory. After being exposed to C#, XNA (see Section 2.2), and UIGE through various tutorials, students started their second project. This group project continued through the remainder of the semester, and was a vehicle for participants to explore their first game engine.

In addition to illustrating the principles of game development, the course's other objective is to see students create two fully functional games by the end of the semester. Each game is unique, so a general list of features required for all games would be ambiguous. However, some of the goals for the course are:

- Each game must have at least one feature complete level.
- Each game must have a demo mode that showcases at least one completed level.
- Each game must have interaction so that there is a definite winning and losing condition.

There were several group projects and one solo project that used UIGE, with students' skill levels ranging from knowledgeable to first time game developers (the latter being the majority). The classroom was a mix of both graduate and undergraduate students. The class consisted of the developers who created the engine, students, and the professor.

2.2. Choosing the Engine's Framework

The course's goal is to introduce established programmers to the principles behind game development. Taking that into account, UIGE needed to be robust enough so that its users didn't feel limited, and also highly structured so that the core components of video game design could be easily taught.

Before it was constructed, UIGE's creators established a few simple criteria the engine's framework needed to meet:

- The engine should use a high-level, versatile language
- The framework for the engine should provide a number of necessary libraries and

functions (game loop, math libraries, graphics device manager, etc.), and allow for easy integration of content into its pipeline

• Other libraries could be easily included to enhance the engine's toolset

After considering various free-to-use options, the DigitalRune Engine [1] was chosen as the platform UIGE would be built on top of. The DigitalRune Engine uses the Microsoft XNA Game Studio 4.0 [2], which includes the XNA Framework 4.0. The XNA Framework provides the basic libraries needed for game development, such as game components and device managers. The DigitalRune Engine expands upon the framework, adding a 3D physics simulation, particle manager, user interface controls, and other useful libraries. These libraries are linked and made usable in the XNA Game Studio, which is a programming environment within Microsoft Visual Studio 2010.

The benefits to using the XNA Game Studio are numerous. XNA allows students to implement their games in a streamlined environment. It uses the self-managed and easy to use programming language, C#. The libraries provided by the XNA Framework contain all the basic necessities used in game development. Because of XNA's widespread use, there is a large user community and tutorials readily available for students. In addition, a lightweight engine built on the XNA framework was used in previous semesters and proved to be a powerful and easily teachable framework to course participants.

The DigitalRune Engine provides a number of additional capabilities. DigitalRune is constructed to be highly modular. One can use all of its libraries in tandem, or pick and choose which ones are needed for their application. Inclusion of libraries from other parties can be easily combined with DigitalRune's various library packages. The DigitalRune Physics package provides a superb 3D simulation, and contains a robust controllable character object. Key-frame animations for sprite sheets and continuous, animatable properties are easily implemented with DigitalRune's Animation package. A variety of useful user interface controls (sub-windows, menus, sliders, etc.) are included with DigitalRune's Game UI module.

For all the reasons listed above, it is easy to see why UIGE was built upon the DigitalRune Game Engine. Additionally, it was prefered that the engine used the C# programming language over a more accessible language, like C++. C# is self-managed and does not allow for direct address manipulation, and although students in the course have advanced knowledge on memory management techniques, they are still prone to one-off errors and faulty memory deallocation mistakes. Debugging such errors tends to take a significant amount of time. Thus, it was determined an engine that used C# would allow its users to concentrate on all other aspects of their applications, and therefore increase productivity.

2.3. Molding an Engine for Students

In addition to the criteria listed in Section 2.1., further standards were established by UIGE's developers to improve the engine's usability and overall productivity:

• The engine should lock objects' movements in the game's simulation to two dimensions. Those objects have 3D bodies, but should have 2D textures (sprites) attached to represent

• A basic game or set of tutorials should be provided for users to expand upon. This foundation should showcase the major components of the engine

• Additional classes and tools (such as a robust camera, object and state managers, level editor, etc.) should be added to reduce the workload on students

Limiting objects to two dimensions within UIGE was done for a several reasons. For one, most students enrolled in the course do not yet posses 3D math, linear algebra, or 3D graphics skills.

Also, 3D objects need 3D models, which require modeling tools to create. Thus, allowing students in the course to work with 3D objects would be a mistake, as a major portion of their development time would have to be diverted to learning these tools and techniques. Participants only have a few weeks to create their games, so time is better spent elsewhere.The desire to provide a basic game for students to expand upon stemmed from past experiences

UIGE's developers had when they were first introduced to video game development. When exploring a large file system such as a game engine for the first time, it is easy to feel lost. It helps to have examples available that guide attention to the major components of the system. Including a basic game or set of linked tutorials is even more desirable, as they can highlight how each component works together to form a fully realized product.

Part of the reason the DigitalRune Engine was chosen as UIGE's foundation is because it offers a number of examples that showcase its various packages. Particularly, the Character Controller example in the Physics package (see section 3.3.1) proved to be a great resource when UIGE's creators were attempting to produce a basic game for the engine. The controllable object in the example had tight controls, fluid movement, and was easily extensible. Hence, this object was integrated into a special type of Object2D (see section 3.2.1.), and then given to course participants through a tutorial. This tutorial and others (Sections 3.3) cover the basic components of UIGE. Additional classes, such as a camera object, object manager, game states, etc. were appended to UIGE so that students would not have to construct their own. The level editor is especially important, because it allows students to quickly create levels (traditionally a tedious task).

2.4. Characteristics of the ULL Introductory Game Engine

A more in depth look at the design process and construction of UIGE can be found in Section 3, but here are a few characteristics of the engine:

- UIGE expands the DigitalRune Engine, so nearly all of the benefits given by the DigitalRune Engine and XNA Framework as listed previously still hold true

- It includes a game object class, Object2D, that allows for easy integration of sprite sheets for animation, a 3D geometric body, distinct types of interaction within the simulation (static, dynamic, kinematic), and various other options

- It comes equipped with an object manager that organizes and handles all Object2Ds in the simulation

- The provided camera class projects the 3D world space into 2D. It can be zoomed in or out, controlled via input, and attached to objects in the simulation

- The state manager and game state classes help organize levels and menus

- A special game state, named the level editor, can be added to and accessed through the state manager. With the editor, one can add objects to a level in real-time, save the level to an XML file, and load the level later as one of the game states

3. ENGINE DEVELOPMENT TIMELINE

3.1. Planning Stage

After the engine's developers had chosen the Digital Rune Engine to be UIGE's foundation, they explored the various packages its framework provided. Digital Rune offers a number of modules that contain analogous libraries. For example the physics namespace encapsulates related libraries such as force effects, simulation, rigid body, etc. together under one roof. UI, animation, mathematics, particles, and physics are some of the packages Digital Rune includes, and each were explored thoroughly when UIGE was being planned. This gave its developers an idea of the tools available, and allowed them to begin forming a general layout for UIGE's architecture (Fig 2.).

After examining what DigitalRune had to offer, UIGE's developers believed a dedicated object manager needed to be included. This would necessitate a wrapper to make DigitalRune's physics libraries compatible with the object manager. Components that could be salvaged from engines previously used in the course, such as sound and alarm managers, should be reused and integrated into UIGE. Most importantly, ease-of-use tools such as a robust level editor and state manager should be constructed. These tools would allow students to rapidly create and deploy levels in their games.

One of the constraints imposed on UIGE was that the engine should only be able to create games locked to a two-dimensional plane. This was an issue, because DigitalRune's physics libraries were designed for three dimensional games. UIGE's developers solved this problem by locking objects within the simulation to two axes of movement, and by creating a specialized object, named Object2D (Section 3.2.1).

In order to have UIGE ready by the start of the students' group projects, its development was separated into the following sequence of milestones (Fig 1):

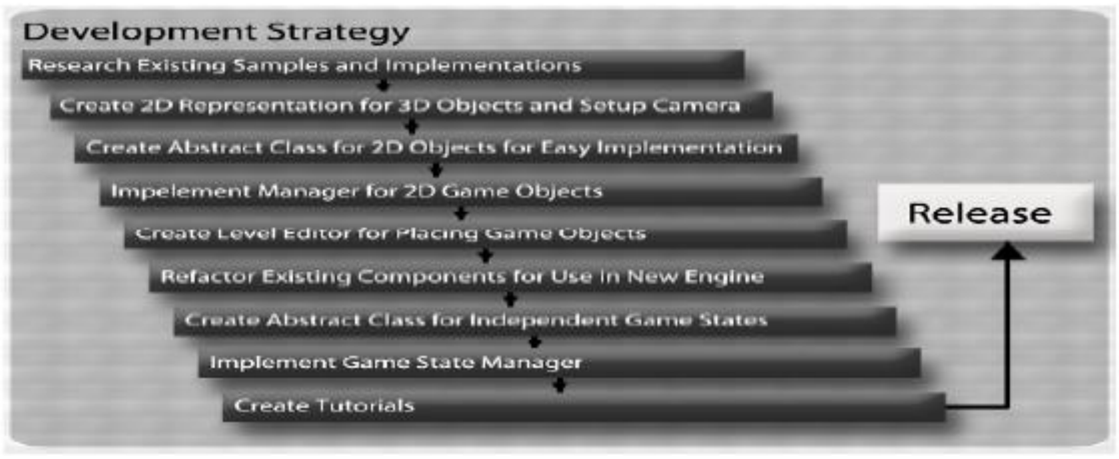

Development Strategy
Research Existing Samples and Implementations
Create 2D Representation for 3D Objects and Setup Camera
Create Abstract Class for 2D Objects for Easy Implementation
Impelement Manager for 2D Game Objects
Create Level Editor for Placing Game Objects
Refactor Existing Components for Use in New Engine
Create Abstract Class for Independent Game States
Implement Game State Manager
Create Tutorials
Release

Figure 1. Development Strategy.

This sequential approach made the creation of the engine within the allotted time possible. Breaking down development into short term goals not only made the project seem more feasible from a developer standpoint, but also illustrates the large project development strategy that is anticipated to be used by students developing with UIGE.

3.2. Implementation

When implementing UIGE, it was a necessity to create an easy to use coding base capable of creating games of multiple 2D genres. Using UIGE should introduce students to common game development methods and practices, such as working with update and draw functions, inheritance, abstract and virtual classes. The engine's design should allow for a great deal of flexibility to facilitate its use in the development of games spanning several genres. After researching the available resources given by DigitalRune and XNA, the development team decided upon the following architecture (Fig 2):

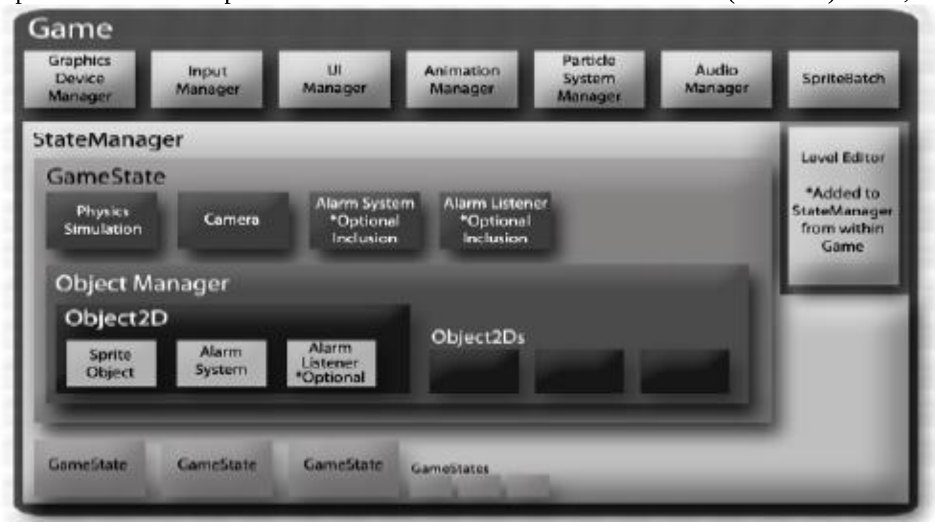


Figure 2. Engine Architecture

This layered approach allows for a great deal of developer freedom while still being manageable and efficient. In the implementation of UIGE, a number of DigitalRune's modules were expanded, while other components were constructed from scratch. The various components of the engine are described with greater detail in the following sections.

3.2.1. ObjectSprite / Object2D / Camera

The first step of implementation was to create a base 2D object that UIGE's users could expand. This was a challenge, since DigitalRune is implemented as a 3D game engine. Its objects needed to be modified in order to be presented properly for two dimensional use. To have DigitalRune's 3D objects display in 2D, a sprite object, named ObjectSprite, was implemented. ObjectSprites generate billboard-like objects that are referred to as a quads. When instantiated, this quad object defines four points in the 3D game world that are aligned with the object being represented, and a 2D texture is projected onto it (Fig 3). Each object of the game world that is displayed creates and maintains an ObjectSprite object.

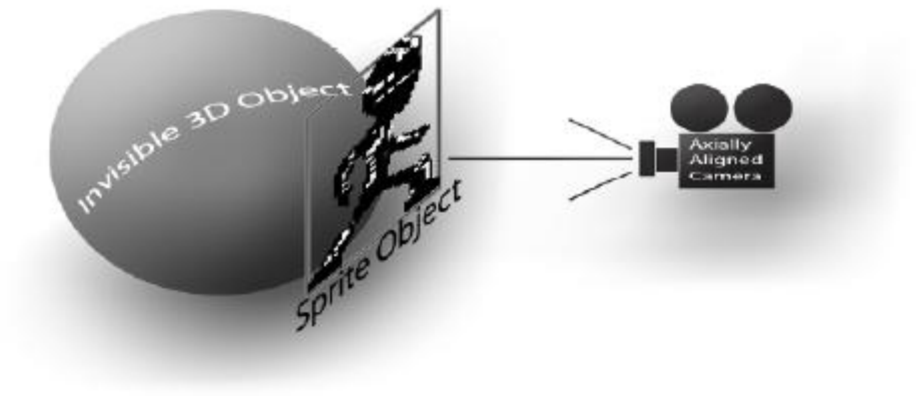


Figure 3. ObjectSprite Representation

Once created and given a height and width, the ObjectSprite generates a quad that can be manipulated by the object. The ObjectSprite contains several manipulatable properties such as tint colors, pose, and the texture that is applied to the quad. It provides several functions to changes these values, as well as resizing tools. ObjectSprites also contain initialization, content loading, updating, and drawing functions that are called by their respective game objects. Each game object can create and maintain multiple ObjectSprites.

In order for UIGE's objects to be compatible with the physics simulation that is used by Digita lRune, three dimensional rigid bodies were used within each force affected object (static, non-collidable objects don't require rigid bodies). In the initialization of an object, the user can pass a shape from the DigitalRune geometry namespace with a specified size. This shape is used when applying any forces, or when detecting collision with other objects.

To ease the learning curve of the engine, an abstract class, named Object2D, was provided that handles some of the common functions of game objects. It contains overridable virtual functions, such as initialization, content loading, update, draw, and cleanup functions. It also contains virtual functions that handle any internal functionality that is required to be compatible with the game engine, as well as any managers that handle the objects.

Every user-created game object that inherits the Object2D class has the ability to override and add functionality to almost any function of the abstract class. This allows users to create dramatically different game objects that are still compatible with UIGE. For example, it is possible to create a static background object that is purely aesthetic, a platformer player character that is controllable via user input and affected by its own gravity, and a top-down style AI driven creature that is interactive and has its own animations. Each object, if implemented correctly, can be created, managed, and destroyed by the game engine. It also allows for the creation and cleanup of child objects within Object2Ds. If Object2Ds are used to their full potential, it is possible to create very dynamic and complex game objects that can be used in interesting ways within the game world, all while being fully managed.

All of the features described above would be rendered pointless if not displayed properly. To effectively display Object2Ds so that they appear two dimensional, DigitalRune's existing camera object had to be modified. The primary adjustment made to the camera was to set its projection matrix to an axially aligned orthographic view; meaning that the camera was set to view down the game world's Z axis and all objects are displayed without foreshortening due to distance of the object from the camera. This sufficed to properly display the objects in the game world. Other modifications were made to the camera strictly for convenience. Included in the updated camera object were methods to lock the camera to an Object2D, follow a defined path, dynamically add waypoints to the camera's path, and animate the camera motion. These additions assisted greatly in making a very believable 2D environment.

3.2.2. Managers

3.2.2.1. Object Manager

The Object Manager handles Object2Ds. The manager is what calls the common methods of the Object2Ds; such as Update, Draw, Cleanup. Managed objects are given a unique identifier and placed within a private dictionary that is only handled by the manager. The object manager also contains lists for objects that are being added to or removed from it. This is done because an object added during a program's execution must be initialized and have its content loaded before it can be updated to avoid crashes and instability. Also, similar steps must be taken when removing an object from the game world to avoid references to an object that is no longer in the manager. The object manager has functions to append objects to the add and destroy lists which can be called at any point in the execution of the game. Then, once the manager reaches its update nfunction it first handles the initialization and loading of newly added objects, gives them each a unique identifier, adds them to the dictionary of game objects, and clears the added objects list since they are now accounted for in the dictionary. Next, all of the objects are updated as normal. Finally, the objects in the destroyed list are disposed of and removed from both the dictionary of game objects and the list of objects to be destroyed. This process is illustrated in Figure 4.

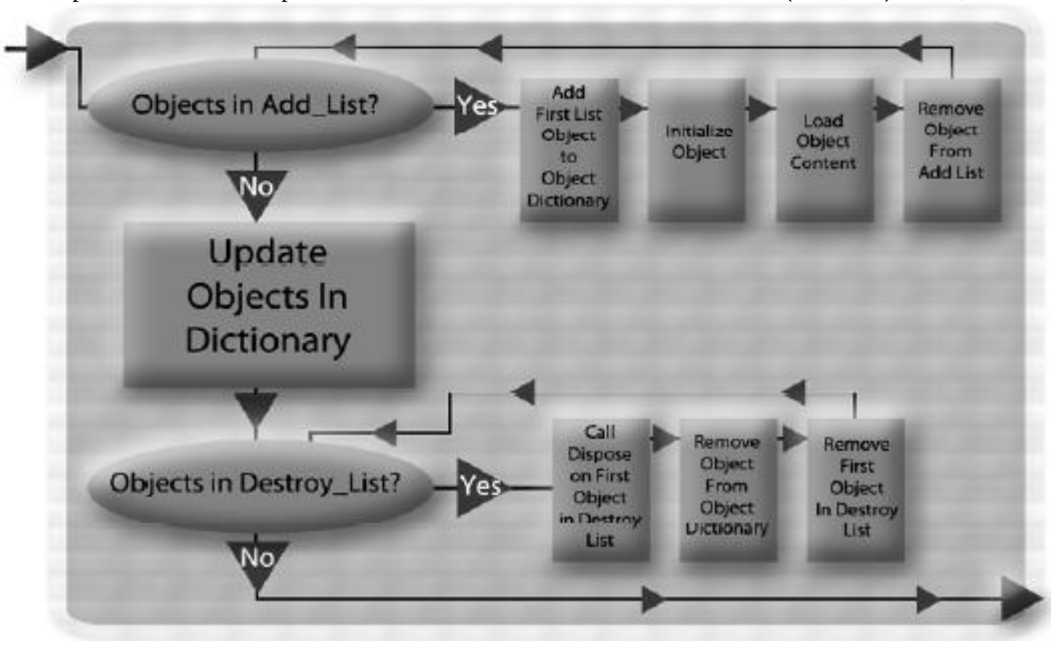

Objects in Add_List?
Yes
No
Add First List Object to Object Dictionary
Initialize Object
Load Object Content
Remove Object From Add List
Update Objects In Dictionary
Objects in Destroy_List?
Yes
No
Call Dispose on First Object in Destroy List
Remove Object From Object Dictionary
Remove First Object In Destroy List

Figure 4. Object Management Flow Chart

Other functions of the object manager include the abilities to easily check for the existence of objects and to be able to pull data from the objects within the dictionary. To do this, the object manager class implements an interface that contains methods for selecting game objects or lists of game objects based on several criteria, such as the object's unique id, the rigid body of the object, the collision body of the object, a type name, etc.

To find the objects that meet the search criteria in the dictionary, the manager queries the dictionary using C#'s Linq library. The System.Linq namespace provides classes and interfaces that support queries that use Language-Integrated Query (LINQ). This avoids manual searching through the dictionary object by object and cleans up the code greatly. The manager makes referencing other game objects a very simple and painless process, and through use of the Linq library, querying objects is efficient enough to happen multiple times per update without significant degradation of performance.

3.2.2.2. Alarm Manager

The alarm manager handles alarm objects. Alarm objects are timers that raise an event whenever their timer expires. Whenever a programmer wants to have an event trigger after a certain amount of time, they create an alarm object, give it an expiration time, and a string message to associate it with an event. When the timer expires, the message included with the timer is sent to the event listener, where any function associated with the particular message will be triggered.

3.2.2.3. Animation Manager

The animation manager is similar to the alarm manager. However, rather than associating specific events with a timer, it updates a value based property over time. A programmer may create an animatable property, decide how the property will change over time, and then push the property into the animation manager. The manager updates the property accordingly and the animatable property can be used as though it were any other variable of its type.

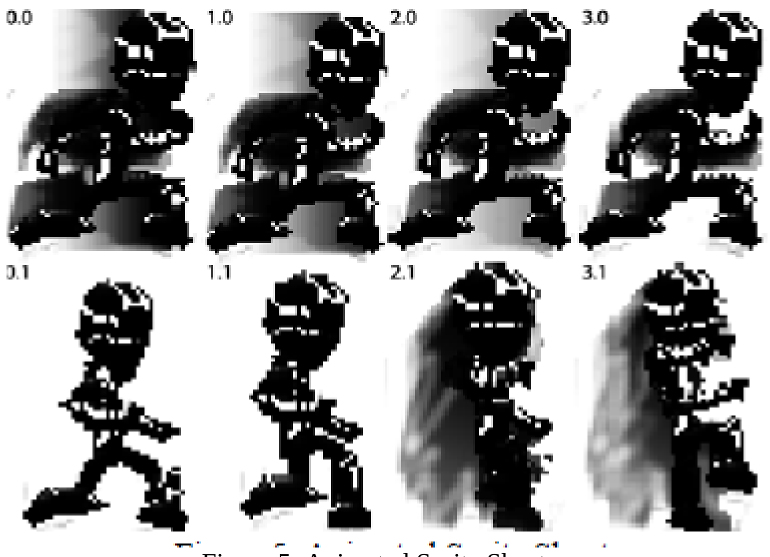


Figure 5. Animated Sprite Sheet

For example, consider an animatable integer property used to animate a game object's sprite. An animation clip is created with a starting and ending value, as well as a duration to preside over. In addition, the clip may loop in a number of ways (constant, cyclic, etc.) and have a blending factor (such as ease-in or ease-out) to smooth out the animation. Once added to the animation manager, the clip will animate the targeted integer property. The property is then used as an index corresponding to a subimage of the sprite sheet (Fig 5). When a game object is created, it pushes the integer property into the manager which updates it automatically rather than having toexplicitly update the subimage index from within the object. When the object is drawn, it will display the currently selected subimage.

3.2.2.4. State Manager

The state manager controls the flow of the game from one state to another, including substates. The manager is inherited from XNA's drawable game component class. This allows it to be added to a game's list of components, meaning that its content loading, update, and draw methods are automatically called as the game progresses without explicitly being called by the programmer. The game states themselves have their methods called from within the manager. This implementation allows for the manager to handle states with different priority levels depending on their type (gamestate, substate, pausestate). As for the game states themselves, the implementation is similar to that of object2D's. An abstract class is provided that contains virtual functions which handle most of the required actions for the manager, as well as references to necessary game components and managers. This abstract class can be inherited and its methods overridden and expanded to meet the developer's needs.

Included in the state manager are multiple private data structures to contain the game states. In the simplest multi-state game implementation, game states can be added to a straightforward list of game states within the manager. The manager handles this list by keeping a reference to the position of the active game state and only calling the methods of that state. Game states can be transitioned by calling methods within the state manager and passing the unique name of the game state. If the game states were added in the appropriate order, they can be cycled through using manager methods for swapping to the next and previous states.

Also included in the manager is a list of substates for a currently active game state. This allows for the inclusion of separate rooms, battles, menus, etc. within the same game state. The list of substates is managed similarly to the list of game states. The index of the active substate is stored within the manager. Notable properties of substates are:

• When a substate is added, the currently active game state is still allowed to update and has the option to continue drawing in the background of the substate, but it loses the focus of the manager

• A state that loses the manager's focus calls a virtual method within the state that sets several variables which determine capabilities of the state; such as the ability to draw its game objects, the ability to update, as well as any other functionality that is defined by the user

• If the current active game state is changed, the list of its substates is cleared before the new game state gains the game's focus

The last data structure included within the manager is a pause stack. If a game state is added to the pause stack, the manager immediately removes focus from active game states and substates and gives the pause state the manager's full attention. This stack is checked before any other state's update is considered. If the manager's focus is on a pause state and the state manager is signaled to change substates, the pause stack is cleared and the new substate gains focus. If a pause state has the manager's focus and the manager is signaled to change the active game state, the pause stack is cleared along with the list of substates and the new game state becomes the manager's focus. A representation of how the state manager prioritizes what state has focus is presented in Figure 6.

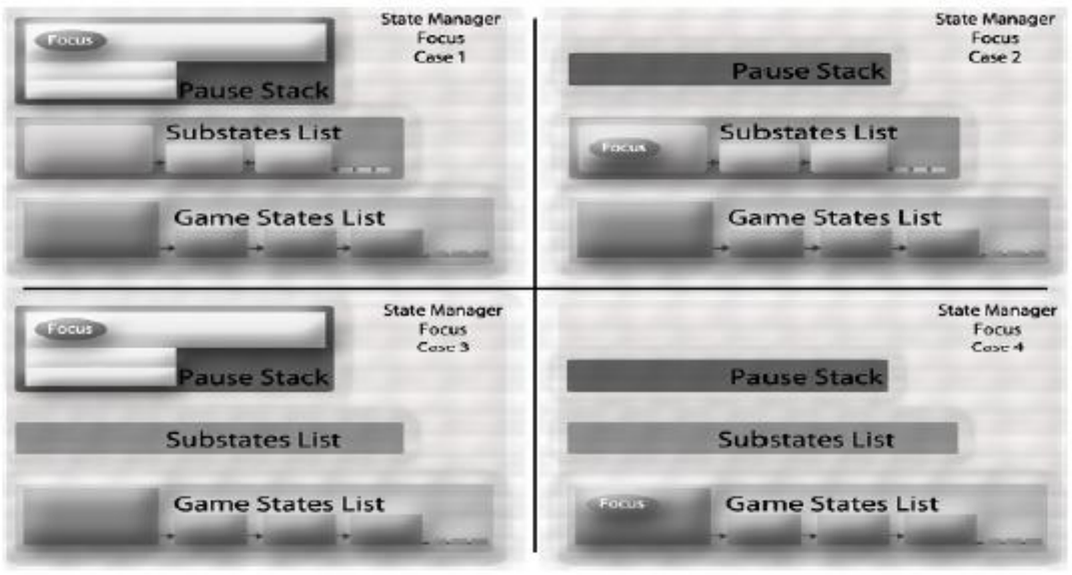


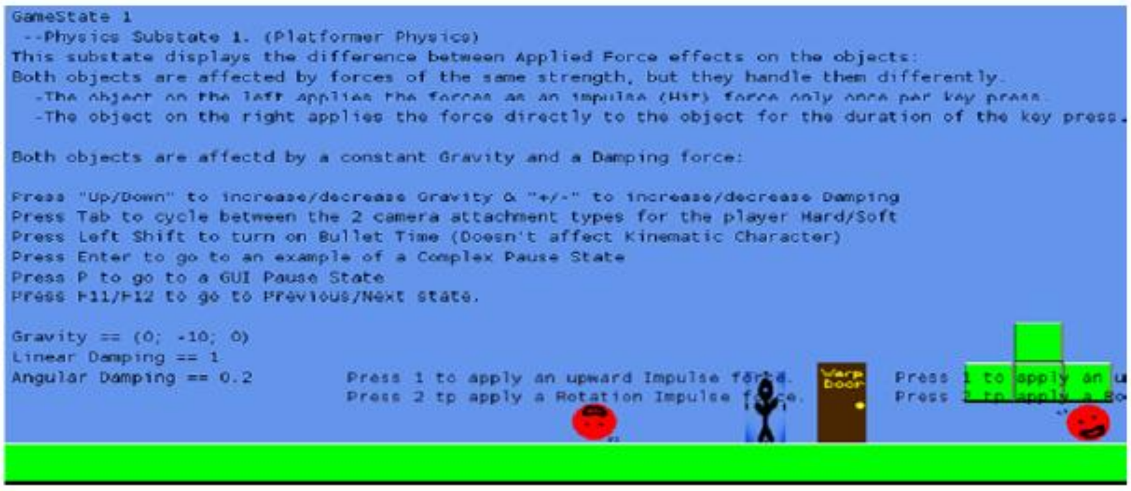
Figure 6. State Manager Focus Examples

3.2.2.5. Built-in Managers

The UI, input, graphics device, and particle system managers are tools built into either DigitalRune or XNA that were not directly modified for students. The UI manager handles interface elements such as buttons, sliders, windows, and the events tied to those controls. The input manager handles keyboard, mouse, and gamepad input events. The graphics device manager handles window and viewport manipulation, such as resolution and full screen mode. The particle system manager handles any particle system added to the game.

3.3. Tutorials

A series of tutorials were created to illustrate the main components of UIGE. Each is described briefly in the sections that follow.



```
GameState 1
 --Physics Substate 1. (Platformer Physics)
This substate displays the difference between Applied Force effects on the objects:
Both objects are affected by forces of the same strength, but they handle them differently.
  -The object on the left applies the forces as an impulse (Hit) force only once per key press.
  -The object on the right applies the force directly to the object for the duration of the key press.

Both objects are affectd by a constant Gravity and a Damping force:

Press "Up/Down" to increase/decrease Gravity & "+/-" to increase/decrease Damping
Press Tab to cycle between the 2 camera attachment types for the player Hard/Soft
Press Left Shift to turn on Bullet Time (Doesn't affect Kinematic Character)
Press Enter to go to an example of a Complex Pause State
Press P to go to a GUI Pause State
Press F11/F12 to go to Previous/Next state.

Gravity == (0; -10; 0)
Linear Damping == 1
Angular Damping == 0.2       Press 1 to apply an upward Impulse force.        Press 1 to apply an u
                             Press 2 tp apply a Rotation Impulse force.       Press 2 tp apply a Ro
```

Figure 7. First Game Objects Tutorial

3.3.1 First Game Objects

The first tutorial that students explore (Fig. 7) instructs them on how to craft dynamic game objects within UIGE. The object they create can move around via player input and interact with other objects in an environment. This tutorial also guides students through the process of creating a sprite sheet for an object, adding the texture to the object, and cycling through sheet to see the object animate on screen. The controllable object given to students in this tutorial was taken from the Character Controller Example in DigitalRune's Physics package. The object can be given a custom kinematic or dynamic physics body. The kinematic body is not affected by force effects present in the simulation but instead applies its own force effects, such as gravity, to itself. Using a kinematic body can make interactions with other objects more customizable. The dynamic body is treated as any other object in the simulation, and as such is less customizable. However, the default kinematic body has some issues with fluid movement, so a dynamic body is a good choice if the user doesn't wish to expand the body's abilities. Groups were encouraged to use and expand this object in their games, especially if they were creating a platforming or top down game. This object, along with the level editor (see 3.4) and other tools, were key components of UIGE's design.

3.3.2 Alarms, Collisions, and Sounds

The second tutorial teaches students the principles of collision detection, how to manage alarm events, and how to add and trigger sound effects and songs with UIGE. These components are sometimes dependant on one another, so grouping them under one tutorial was an easy choice. For example, consider a missile object colliding with another object. The missile explodes, which requires an explosion sound to be played. In addition, the other object takes damage. An alarm in the form of a cooldown may then be triggered so that the object won't take additional damage for some short period of time.

3.3.3 States and User Interface Controls

The third tutorial shows students how to create and link game states. The example also introduces participants to user interfaces and controls, and demonstrates how to add them to game states. Students were required to implement buttons that traversed through states in the state machine when clicked. In addition, they studied the implementation of a pause state and adjusted menu controls that were included with it.

3.4. Level Editor

Increasing development speed with UIGE had always been its developers first priority. After implementing the engine’s core components, UIGE’s developers deemed the addition of a real time editor to be the optimal way to improve throughput. If one were to look at the most popular game engines used today by both professionals and amateurs, the common feature they’d find between them is the inclusion of a robust level editing tool. See Unreal Game Editor [4] and the Unity Editor [5].

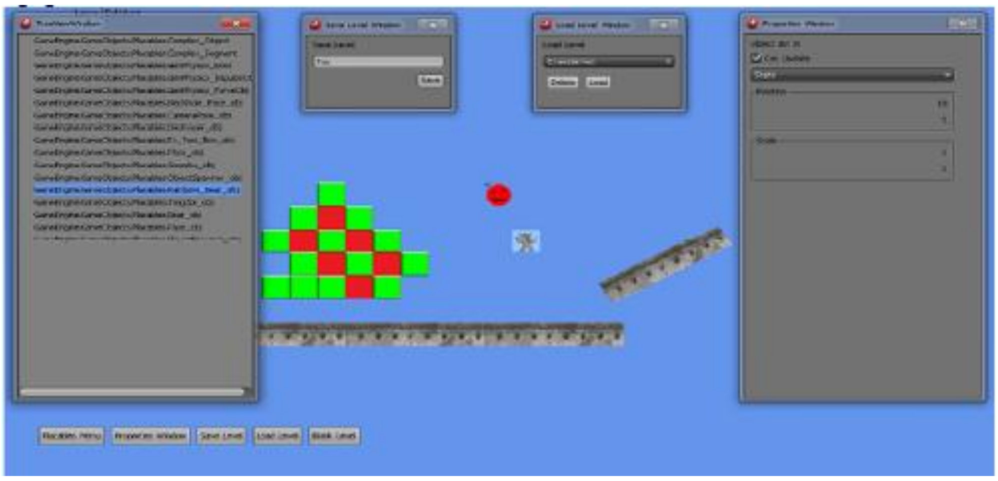
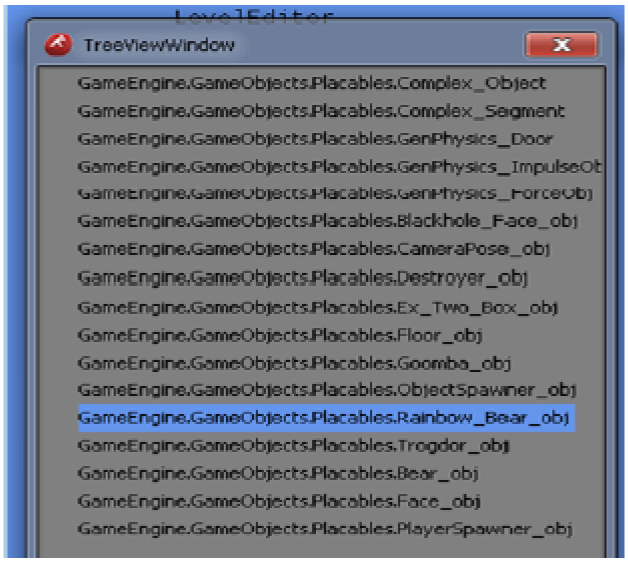


Figure 8. Level Editor



LevelEditor

TreeViewWindow

```
GameEngine.GameObjects.Placables.Complex_Object
GameEngine.GameObjects.Placables.Complex_Segment
GameEngine.GameObjects.Placables.GenPhysics_Door
GameEngine.GameObjects.Placables.GenPhysics_ImpulseOb
GameEngine.GameObjects.Placables.GenPhysics_ForceObj
GameEngine.GameObjects.Placables.Blackhole_Face_obj
GameEngine.GameObjects.Placables.CameraPose_obj
GameEngine.GameObjects.Placables.Destroyer_obj
GameEngine.GameObjects.Placables.Ex_Two_Box_obj
GameEngine.GameObjects.Placables.Floor_obj
GameEngine.GameObjects.Placables.Goomba_obj
GameEngine.GameObjects.Placables.ObjectSpawner_obj
GameEngine.GameObjects.Placables.Rainbow_Bear_obj
GameEngine.GameObjects.Placables.Trogdor_obj
GameEngine.GameObjects.Placables.Bear_obj
GameEngine.GameObjects.Placables.Face_obj
GameEngine.GameObjects.Placables.PlayerSpawner_obj
```

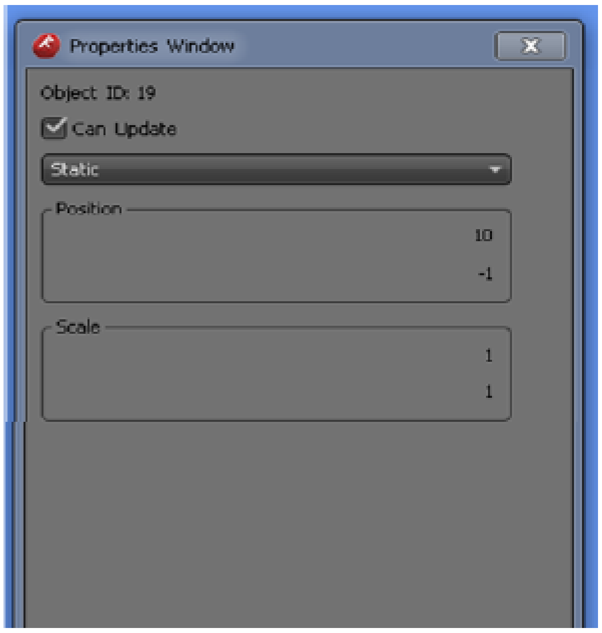
Figure 8a. Level Editor Placeables Window

The benefits of UIGE’s level editor include:

● Allows users to quickly create environments and view them in real time, with no hard coding necessary

● Can quickly test the functionality of objects in a number of scenarios, again in real time

● C# offers extensive support for reflection. Developers can append their objects to a singular namespace and have the editor instantiate the objects automatically at runtime. The list in Figure 8a. contains all the objects under the namespace, Placeables. Clicking on an entity in the list will create an instance of that object



Object ID: 19

☑ Can Update

Static

Position

10

-1

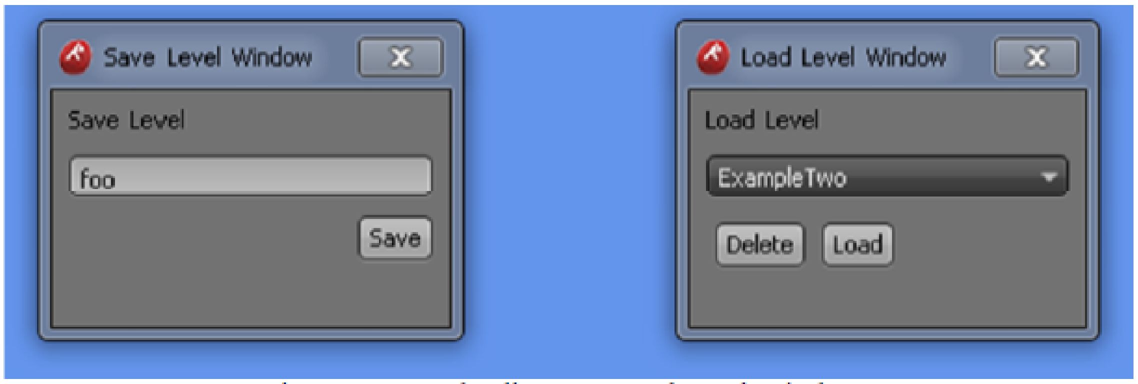
Scale

1

1

Figure 8b. Level Editor Properties Window

The ability to place and test objects in a real time simulation is a useful tool for visualization and debugging purposes, but the experience can be improved with added functionality. UIGE's developers wanted the editor to have the ability to manipulate key attributes of objects placed in the world, such as position, scale, and rotation. A number of input controls were added to manufacture such modifications, and a properties window (see table in Figure 8b.) was included to give users control over objects' spawning options.



foo

ExampleTwo

Figure 8c. Level Editor Save and Load Windows

Of course adding and manipulating objects in real time is pointless if levels cannot be saved and reloaded. These options are included with UIGE's editor and can be seen in Figure 8c. C# provides a serializer that can store properties of objects into an XML file format, which can then be loaded at a later time. Spawners, which spawn objects after some criteria is met, may also be saved using the editor.

The editor provided with UIGE is robust, but there are number of potential enhancements. Currently, the editor supports the manipulation of objects placed within its world space, but any modifications made are to properties common to all objects (such as position and scale). Consider an elevator object. This object has a starting and ending position (offset by its world position) that it animates to and from when the player activates it. Unless the user wants all of their elevators to be displaced by the same amount when activated, they would have to hardcode these properties. Also, they wouldn't be able to place the elevators into world using the editor because the editor can't save the properties (it won't let them be edited either). A solution that allows for the alteration of properties specific to each object would be more desirable, as it would spare users the need to hardcode any of their objects!

4. STUDENT PROJECTS

4.1. Development Timeline

4.1.1. Game Maker Project

Upon entering the course, students are introduced to basic game logic (update and draw loops) m through examples created with GameMaker. They are then given the task of constructing a game proposal document for their first game. The document illustrates the student's ideas and goals they have for their games, and also serves as a guideline for grading once projects are submitted. A proposal can be rejected if administrators deem the plan either too simple or too ambitious. Once the game proposal is completed and approved, students have approximately three weeks to construct their GameMaker games. During this time, course administrators will check on each mstudent's progress and assist them if needed. After the students finish their games, they are required to demonstrate their results to the class. Other classmates will offer criticism (positive and negative) during the presentation and then again with the submission of a peer review document.

4.1.2. XNA/C#/DigitalRune Tutorials

After completing their GameMaker games, students are introduced to C# and Microsoft's XNA Game Studio through a series of short lectures. Participants are then given access to UIGE and tasked with completing three tutorials (described in section 3.3). The examples provided by Digital Rune are also made available for optional viewing. The DigitalRune simulations and guides are not all applicable with UIGE (some are 3D specific, whereas UIGE is 2D), but each of them takes a thorough look into the engine's architecture.

4.1.3. Group Project, Milestone One (The Gray Box)

Once participants are more comfortable with C# and UIGE, they are partitioned into teams of three or four and tasked with creating a game proposal document (similar to the one created for their GameMaker games). Larger groups aren't allowed, as that would limit the range of applications created and potentially overwhelm students with group management issues. After the document is submitted, it is reviewed so that the first milestone of the project can be established. Three to four weeks are spent working towards the milestone.

In addition to specific implementation goals, each project must have a functioning gray box completed by the first milestone's deadline. A gray box, in this case, is a real-time simulation showcasing the core systems of the game without the inclusion of any bells or whistles (such as art assets). For example, the gray box for a platformer game might consist of a controllable player character that can move and jump around in a basic world space without clipping.

Students are also instructed in version control techniques during this time via Subversion (SVN) [6], an open source version control software. These techniques, in conjunction with the group project, teach students how to function in a larger group environment (where more than one teammate may be submitting adjustments to the project simultaneously).

4.1.4. Group Project, Milestone Two

After the gray box has been submitted, each group presents their partially finished game for review.

The projects are critiqued by the rest of the class, who offer advice on which features to scrap, which to keep implementing, etc. Using that feedback, groups then construct a final milestone. Typically, the second half of development will see rapid improvements and additions, as students are better adjusted to using UIGE. This phase lasts four weeks, and concludes with a final submission of the game's executable and game assets.

4.1.5. Final Presentation

On the day of the final, the groups are each given five to ten minutes to present a live demonstration of their (hopefully) completed games. Each member of the team discusses their role and contribution to the project, and are encouraged to provide feedback pertaining to their experiences using UIGE. Other groups may offer any final criticisms and then conclude by submitting a game evaluation document.

4.2. Games Developed Using UIGE

Here are a few of the games created with UIGE.

4.2.1. Oncomon

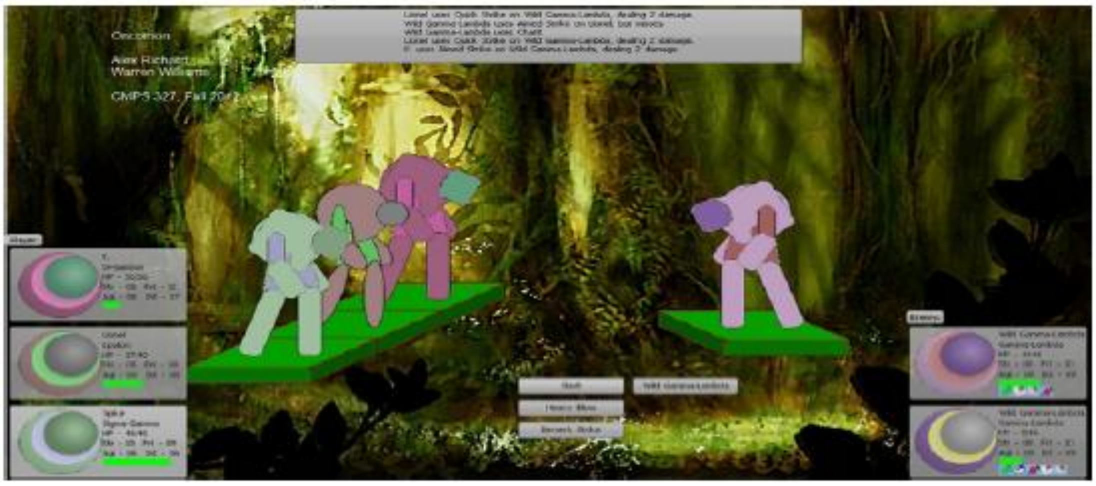


Figure 9. Oncomon Gameplay Screenshot

Oncomon is a menu based monster breeder and battle game. Players are given a set of monsters which battle other monsters in the wild in order to collect their DNA. The genetic material can be spliced together to create new, more powerful creatures. The goal of the game is to create the perfect monster, which can only be crafted after collecting and experimenting with a substantial amount of DNA.

The game makes heavy use of DigitalRune's UI package. Everything in the game is controlled via UI controls, and the only input required is the left mouse button. Oncomon makes up for the lack of quick, reflex based combat by providing players with truly deep game systems. Oncomon surprised UIGE's developers because they were not expecting to see students veer far from the platforming genre with their games. In that sense, Oncomon is a good example of the range of applications UIGE supports.

4.2.2. Portal Hoppers

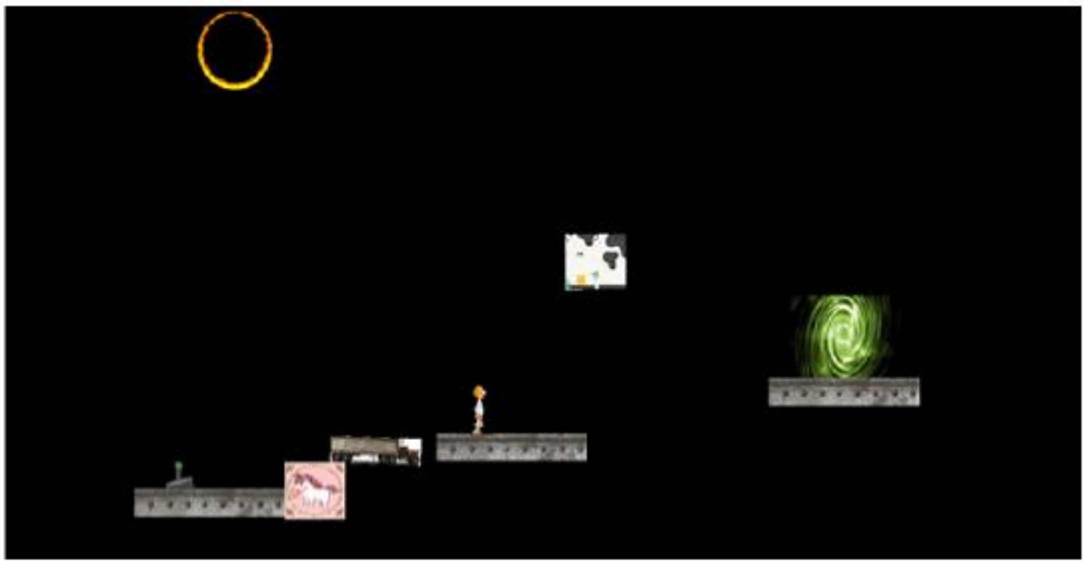


Figure 10. Portal Hoppers Gameplay Screenshot

Portal Hoppers (Fig 10) is a platforming game with a twist. The playable character, Paul, is lost in space and needs to traverse a series of portals to return to his home dimension. Paul needs the assistance of his object spawning ability to safely make it to each portal. Once an object has been spawned into the world, Paul can connect it to other objects in order to create a path to the portals. Portal Hoppers takes some of the techniques used in UIGE's level editor to spawn and manipulate objects. Players can manipulate spawned objects in a variety of ways via input controls.

4.2.3. Hubris

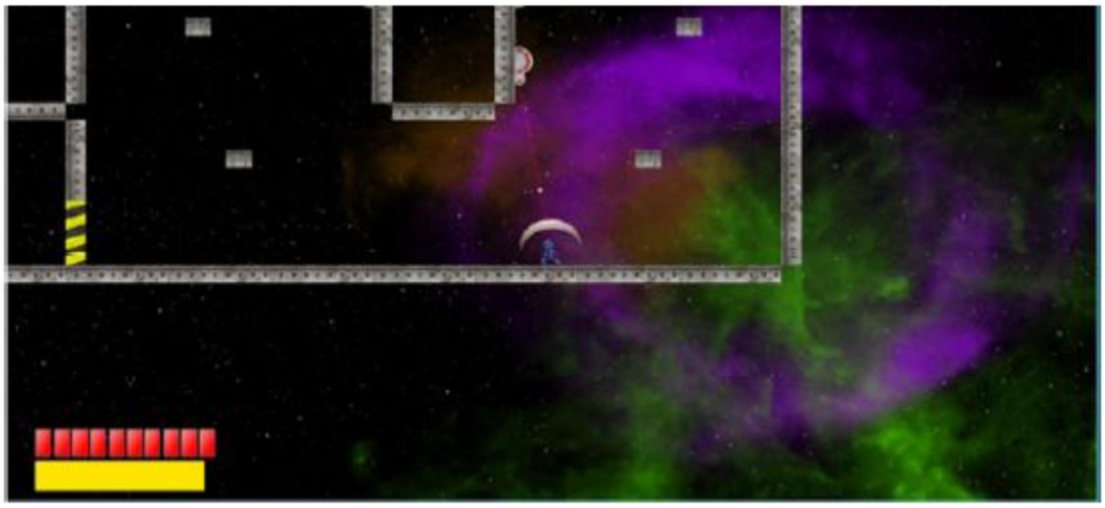


Figure 11. Hubris Gameplay

Hubris [7] is a sidescrolling platformer created by one of UIGE's developers The project was a solo effort on his part; all code and assets were created by a single programmer with the exception of the playable character's sprite sheet. The game was constructed in six weeks. Hubris consists of five levels, and the goal is to make it to the portal at the end of each stage. It takes advantage of a number of UIGE’s components, such as the level editor, state machine, and collision event functions provided by Object2Ds. The ray tracer and particle systems (see bullet in Fig. 11) included with DigitalRune are used by a number of the game’s objects.

5. RELATED WORK

Masuch and Rueger [8] discuss teaching game design at a university level. They describe how they balance their curriculum to ensure their students are not overwhelmed. They take care to m expose their students to a wide range of tools and systems to broaden the skills gained while also being careful not to delve so deep into a specific topic that the student becomes lost in the detail. The authors also discuss the differences between a professional and educational game development environment in terms of resources with regard to time, experience, and budget. The authors also describe their past experiences in the classroom with different game development environments ranging from GameMaker to the game engine that they used, Impara. The systems included are discussed including examples of how they were used in the games created by the students.

Ryoo, Fonseca, and Janzen [9] utilize game design theory in order to teach object oriented programming (OOP) and software engineering. They explain that the difficulty in teaching object oriented programming to inexperienced students can be lessened by making the experience more hands on. Some of the challenges faced in teaching OOP are similar to the topics discussed by Masuch and Rueger such as fighting time constraints and ensuring to not overemphasize a minute facet of a concept and instead maintaining a broad focus on a variety of topics. The solution examined is the breaking up of course into four phases: inception, elaboration, construction, and transition. Boudreaux, Kumar, and Etheredge [10] explain the steps to take a game from concept to product and the tools used to do so. The authors discuss the game design course and the steps taken by the students to arrive at a completed game. The course is divided into multiple milestones beginning with a 2D project in GameMaker to grasp the basic concepts of programming a game. The next project tasks the students with creating a full 3D game in OGRE. The article also breaks down an individual project into the pieces that compose the created game.

Shabanah, et al, in [11] describe designing computer games in order to teach algorithms. They first explain how teaching algorithms is difficult because they are usually applied to complex problems such as variances in data structures, or complicated mathematical theories. To alleviate this difficulty, they propose the use of games to simplify the learning process allowing students to learn about algorithms on a more friendly level. They accomplish this by juxtaposing algorithm visualizations and games which makes learning the algorithms easier to understand and implement.

Mark Overmars in [12] discusses tools available to developers used to make games. He also considers how to use these tools in order to teach computer science. He assesses GameMaker and walks through certain facets of its user interface and also describes its usefulness in teaching object oriented programming in a similar fashion to Ryoo, Fonseca, and Janzen from [9].

6. CONCLUSIONS

The objective of the course is twofold -- To impart the principles of video game design and development to its participants, and to have students see their projects through to fruition. Striking a balance between these goals has always been a difficult challenge for course administrators. Past versions of the course tended to adhere to one goal while sacrificing the other. For example, GameMaker: Studio had been used in past semesters to create both the solo and group projects. This allowed students to successfully produce finished products; however, the abstract nature of GameMaker’s scripting language resulted in a lack of exposure to the interworking systems under the hood. On the other end of the spectrum, more recent editions of the course had participants using a more bare bones game engine. It was a great tool for illustrating the core components of game development, but a majority of groups who used the engine failed to present a completed game by their project's deadline.

UIGE was designed to achieve a reasonable balance between the two goals. It provides its users with a variety of tools and a strong foundation to expand upon, while still making available all of the engine's components. The group projects constructed in the Fall 2012 edition of the course illustrate UIGE's strength as an engine and teaching device. The games created were some of the best to come out of the course, and students praised the complexity UIGE had over GameMaker.

The engine is not without its faults, of course. The Fall semester of 2012 had a particularly large amount of graduate students taking the class, and their shared criticism was that UIGE was too restrictive in the range of applications it could support (efficiently). Because the engine requires there to be a physics simulation added to each game state, users who do not use rigid bodies for their objects, or need use of physics in general, are forced to have an impractical component in their levels. Many of the tools added to UIGE also rely on the simulation (such as collision detection), and to have them function without a simulation would require modification on the student's part. A rebuttal to these criticisms is that UIGE is not meant for advanced users, and hat the loss of efficiency shouldn't impact the quality of any of the games created within the scope of the course (most games created in the course are fairly resource light).

Having students craft a fully fleshed out game in six to seven weeks, while still allowing them the full catalog of tools available to a game engine, is unrealistic. UGIE was crafted by students who had inherited the experiences of engine development from previous administrators of the course. Continued development of the engine or new engines in the future will benefit from knowledge aggregated through the iterative construction of tools like UIGE. This paper will provide future instructors of the course insight into the development of such tools, and should give instructors of similar programs a reference to use when constructing their own introductory to video game development courses.

REFERENCES

[1] GameMaker: Studio. http://www.yoyogames.com/gamemaker/studio

[2] DigitalRune. http://www.digitalrune.com/

[3] Microsoft XNA Game Studio. http://msdn.microsoft.com/en-us/centrum-xna.aspx

[4] Unreal Game Editor: http://www.unrealengine.com/features/editor/

[5] Unity Editor: http://unity3d.com/unity/workflow/integrated-editor

[6] Apache Subversion. http://subversion.apache.org/

[7] Hubris: https://sites.google.com/site/robertmspearsiii/c/senior-project---hubris

[8] Maic Masuch and Michael Rueger. Challenges in Collaborative Game Design Developing Learning Environments for Creating Games. In Creating, Connecting and Collaborating through Computing, pages 67-74, Los Alamitos, CA, USA, 2005.

[9] Jungwoo Ryoo, Frederico Fonseca, and David Janzen. Teaching Object-Oriented Software Engineering through Problem-Based Learning in the Context of Game Design. In Software Engineering Education and Training, pages 137-144, Altoona, PA, USA, April 2008

[10] Hollie Boudreaux, Ashok Kumar, and Jim Etheredge. An Algorithmic and Software Engineering Based Approach to Robust Video Game Design. In International Journal of Software Engineering &

Application (IJSEA), Vol. 2, pages 35-48, July 2011.

[11] Sahar Shabanah, Jim Chen, Harry Wechsler, Daniel Carr, and Edward Wegman. Designing Computer Games to Teach Algorithms. In Information Technology: New Generations (ITNG), pages 1119-1126, Fairfax, VA, USA, April 2010.

[12] Mark Overmars. Teaching Computer Science through Game Design. In Computer Vol. 37, Issue 4, pages 81-83, Netherlands, April 2004.