GEDI: A GAME ENGINE FOR TEACHING VIDEOGAME DESIGN AND PROGRAMMING

Ron Coleman, Stefen Roebke, Larissa Grayson
Computer Science Department
Marist College
Poughkeepsie, New York 12601
ron.coleman, stefen.roebke, larissa.grayson {@marist.edu}

ABSTRACT
We introduce Gedi, an open source game engine we maintain and support for teaching videogame design and programming in C++.

INTRODUCTION
Two years ago, we created two elective courses, Game Design and Programming (GDP) I and II. They are part of an undergraduate Computer Science concentration in videogame development modeled on The Curriculum Framework: The Study of Games and Game Development. [8] Both courses are project-oriented. That is, in a project setting students first analyze classic 2D games as design references. [9, 13, 15] The references include role-playing, sports, racing, shooter, puzzle, etc., games. Students then program variations of these game designs in C++ on the PC.

We pursued a modern approach to game development based on a game engine. While features vary widely, game engine software generally automates game-specific functions like loading and rendering animations, playing digital sounds, controlling input devices like joysticks, and simulating physics-based special effects. Game engines also lend themselves to applications outside of games, namely, scientific research [12] and feature-film special effects [10]. However, we only required an engine suitable for studying basic principles.

Thus, we searched the field of C++ game engines and found so many that we cannot give a full account of them here. In summary, we tested or reviewed ones like Aurora [2], Torque [16], and CrystalSpace [6]. Unique in their own way, they cover the spectrum of
the expensive, affordable, and open source, respectively. They are also typical, designed for ambitious and/or commercial game development. Hence, their strengths—the versatility and sophistication—are also limitations as the attending complexity can be a distraction for teaching basic principles and a barrier to mastery for newcomers working on a semester life cycle basis.

GEDI

We eventually came across an out-of-print text [5], which develops a C++ game engine called, Mirus. The concept appealed to us because Mirus was specifically designed to create 2D games—just the kind we had in mind. Moreover, on closer inspection, Mirus seemed beautifully conceived in about 5,000 source lines. In other words, Mirus was small enough to be mastered for writing games in a semester. Mirus's compactness moreover seemed to invite unencumbered exploration of its internals as a case study in game engine design. This latter point was not a requirement of our search. Yet, it helped make Mirus an obvious choice.

To start, there was just one small problem. Mirus didn't work. The highlight of Teixeira de Sousa (2002), a pre-compiled game, Breakthrough [8], failed to run. The Mirus class library didn't compile. By play testing [13] our games, we uncovered a series of Mirus run-time bugs and realized Mirus was incomplete. For instance, Mirus has a rigid body physics engine. Breakthrough depends on it to simulate the Newtonian mechanics of ball and paddle interactions in conjunction with Mirus's collision detection algorithms. However, Mirus lacked its own random number generator, which is crucial for creating novelty and challenges in games.

We tried contacting Teixeira de Sousa to no avail. We made several inquiries with the editor and publisher. They told us they, too, were at a loss as the author was missing, their emails unanswered and royalty checks returned.

We understood then that we were on our own, so to speak, that we would have to rescue Mirus ourselves. Therefore, we converted our workarounds to permanent fixes and began extending Mirus. As an example, we added interfaces to popular game design tools like Mappy [11], a level editor for creating "levels" or game worlds. Our goal was to create a reliable teaching platform while preserving Mirus's spirit, namely, its simplicity and patterns, which had attracted us in the first place.

We call the project Game Engine Design and Interface—or Gedi (pronounced, jed-eye). It includes the overhauled and play tested game engine: 10,000 source lines. The project also has a collection of game patterns we built to accelerate game development, demonstration game sources, and complete application program interface (API) documentation, which is missing in Mirus. There are libraries and projects for Dev-C++, a GNU compiler, and Microsoft(r) VC++ .NET. We have made the Gedi project available through the open source community at http://gedi.sourceforge.net to encourage teaching and exploration of game development from a computer science perspective.

The remainder of this paper describes Gedi. We also discuss teaching approaches and experiences using Gedi in GDP I and II.
GEDI BUILDING BLOCKS

From a building block perspective, Gedi consists of the Mirus base from Teixeira de Sousa (2002), our fixes, and our extensions. The extensions take three forms consistent with the goals mentioned above: (i) new methods on the Mirus base classes; (ii) new classes that extend the Mirus base classes through inheritance; and (iii) new classes that encapsulate the Mirus base classes and extended classes.

The point of the figure above is that Gedi relies on Microsoft® Direct3D® and Windows. Yet, Gedi does not generally expose these APIs. In fact, this is one of the primary benefits of a game engine: to abstract away such details and allow programmers to focus on implementing the game design, that is, the level design, interface design, play rules, etc. Gedi uses a façade or wrapper pattern to initialize, access, and generally manage the underlying APIs.

GEDI BASE

Gedi is organized in components or logical groupings of C++ classes. See Table 1. (The "Fixes" column lists the number of bugs we fixed in the respective base component.)

Table 1. Gedi component architecture.

<table>
<thead>
<tr>
<th>Component</th>
<th>Base API classes</th>
<th>Fixes</th>
<th>Extensions API classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helper</td>
<td>Type, error code, and timer</td>
<td>0</td>
<td>Exception and alarm clock</td>
</tr>
<tr>
<td>Window</td>
<td>Window interface</td>
<td>0</td>
<td>Message box class.</td>
</tr>
<tr>
<td>Graphics</td>
<td>Graphics and animation</td>
<td>7</td>
<td>Font, tile map classes</td>
</tr>
<tr>
<td>Input</td>
<td>Keyboard, mouse, and joystick devices</td>
<td>4</td>
<td>State-based classes for non-buffered control.</td>
</tr>
<tr>
<td>Sound</td>
<td>Sound WAV format</td>
<td>1</td>
<td>Sound information and 22 non-WAV formats.</td>
</tr>
<tr>
<td>Math</td>
<td>Matrix, 2D vector classes</td>
<td>0</td>
<td>Uniform and Gaussian deviates.</td>
</tr>
<tr>
<td>Physics</td>
<td>Rigid body physics and particle systems</td>
<td>2</td>
<td>Reusable rigid body physics.</td>
</tr>
</tbody>
</table>

Gedi inherited this architecture from the Mirus base. A notable benefit is that these components provide a highly reusable structure in which to add new classes. Furthermore,
many classes across different components in the Mirus base have the same methods with identical signatures. For instance, Update, Render, and Init are recurring methods that help to simplify class interfaces, making them regular and mnemonic. They are also reusable patterns for writing higher-level game code as we explain below.

Finally, the Mirus base uses prefixed variables. We continue this style in our code examples in this paper. Thus, the prefix m_ denotes class member. The prefix, k, means class instance; i, integer type; p, pointer type; and a, array type.

Window Component

The Window Component is the logically highest level of Gedi. It consists of one class, Window, an abstract base class. Game code must extend Window and implement the Frame method. A game creates an instance of the derived Window class then invokes an inherited method called, Run, which is a forever-loop called the game loop. The structure of the game loop is organized as follows:

```c++
while(true) {
    m_kGame.Update();
    m_kGame.Render();
}
```

The object, m_kGame, is an instance of Game and the top-level container of all other game objects. Unlike Window, the Game class is a part of game code. However, as a game programming convention, Game separates logically distinct game operations: namely, updating and rendering game objects. The Update method of Game executes the following canonical game operations:

```c++
void Game::Update( ) {
    ProcessPlayerInput( );
    SimulatePhysics( );
    DoAI( );
    DetectCollisions( );
    HandleCollisions( );
    UpdateGameState( );
}
```

To process input from the player, for instance, games can use the Joystick class. First, the game code invokes Update on m_kJoystick, an instance of Joystick. The Update method polls the game controller and buffers device data. To retrieve the analog X-axis data and send the result to another game object, say to update or move a Space Invaders laser base, the game code invokesGetXAxis on m_kJoystick:

```c++
m_kJoystick.Update( );
Int32 iXOffset = m_kJoystick.GetXAxis( );
m_kLaserBase.Move(iXOffset);
```

Gedi handles keyboard and mouse devices similarly and directly supports each of the other canonical game operations. The exceptions are artificial intelligence (AI) and game state machines, topics outside the scope of this paper. We note, however, that game state machines are essentially finite state automata (FSA) applied to implementing play
rules, scoring, special effects, etc. [6] For readers interested in AI for games, we recommend Bourge and Seeman (2004).

**Graphics Component**

The animated blittable object or ABO class in the Graphics Component is probably the most important class in Gedi. ABO is primarily responsible for rendering sprites or animated graphics. The Render method of Game, by convention, carries out these operations, as we suggest further below.

The ABO also implements two collision detection algorithms. One operates on rectangular areas and the other, spherical areas. In the rectangular case, two ABO objects collide if their bounding rectangles intersect. In spherical case, they collide if the distance between the bounding spheres is less than the sum of the radii.

For instance, in the above figure, the two ABO objects, P and Q (left) collide if using bounding rectangles since P and Q rectangles intersect. The same objects using bounding spheres do not collide since \( d > r_1 + r_2 \).

In code, to detect whether an alien photon shell has hit a laser base during the update phase of the game loop, games invoke the Collide method, the default of which uses bounding rectangles:

```cpp
If(m_kLaserBaseABO.Collide(m_kPhotonShellABO)) {
    m_kLaserBaseABO.Update( );
    m_iBasesRemaining--;  
}
```

The Update method on the ABO class causes the next frame in the animation to be selected but without rendering it. The SetPosition method of ABO is similar: it positions the ABO on the screen without rendering the ABO. The ABO class has a number of other methods for changing the ABO size, orientation, transparency, and color channels. The color channels affect how the ABO blends with the game world background using programmable color schemes.

By convention, when Game::Update returns to the caller, all ABO objects know (i) where they are in the game world and (ii) how they render. They are only awaiting Game::Render to invoke Render which "blits" (i.e., copies) each one to the video buffer for display:

```cpp
void Game::Render( ) {
    for(Int32 iIndex=0; iIndex < m_iNumberABOs; iIndex++)
        m_pABO[iIndex]->Render( );
}
```
Input and Sound Components

Input and Sound Components are very similar. The InputManager class initializes the system hardware for access by Keyboard, Mouse, and Joystick classes. Their jobs are to acquire and ultimately retrieve device data. On each cycle of the game loop, the game code invokes the respective Update method on instances of these objects to poll the hardware for input. To retrieve the input, they invoke, IsButtonUp or IsButtonDown with a key code as the actual parameter. Joystick objects furthermore have access to GetXAxis and/or GetYAxis, which return the analog axis position.

The Sound Component works similarly. The SoundPlayer class initializes the system sound card for access by Sound objects. A Sound object loads a .WAV file and plays the sound when the game code invokes the Play method.

Physics and Math Components

The Physics Component depends on the Math Component mainly for a class, Vector2D. It implements a number of operations useful for physics simulations like vector arithmetic, normalization, dot product, etc. The Physics Component proper is composed of two distinct types of classes.

The class, Entity, embodies the rigid body physics simulator. [6] It assumes conservation of momentum and idealized interactions, i.e., no deformations, release of energy, etc., among a few Entity objects. The Entity class has methods for manipulating mass, velocity, and acceleration, and for applying forces like friction and gravity in accordance with Newton's laws of motion.

If two Entity objects interact, that is if they collide, the Entity class has a method, HandleCollision, to affect the velocity vector of each Entity. For instance, suppose there are two entities, P and Q corresponding to the ABO objects in Figure 2. The code snippet below tests if P and Q ABO objects collide, computes the effects on P and Q entities, and finally simulates the effect at the next time step, Δt.

```
If(m_kABOP.Collide(m_kABOQ))
  m_kEntityP.HandleCollision(m_kEntityP - m_kEntityQ);
  m_kEntityP.Simulate(fStep);
  m_kEntityQ.Simulate(fStep);
```

HandleCollision updates the velocity vectors, $V_P$ and $V_Q$ using Newton's impulse method [6] and the collision normal vector, $N$. The collision normal vector is the force perpendicular to the line of collision. For spheres, $N$ is the difference between the vector positions of each entity, $S_P$ and $S_Q$. [6]

After a possible interaction, the simulated position information must be transferred to the ABO objects for eventual rendering. Thus, the next code snippet (following the one above) might look like the following:

```
m_kABOP.SetPosition(m_kEntityP.GetPosition( ));
```

![Figure 3. Interaction between P and Q entities at the (dashed) line of collision.](image)
m_kABOQ.SetPosition(m_kEntityQ.GetPosition( ));

The Physics Component also simulates particle systems. They are large collections of "weakly interacting" objects called *particles*. [6]

Each particle, j, follows simple rules: namely, update its color, size, and position as a function of time, t. Each particle has the same lifespan, L, and starts from the same vector position, S. Each particle is also unique with randomized initial conditions of speed and direction, \( V_j \). In the figure above, \( \theta = r_j \) \* \([-\pi, \pi]\) \* D where \( r_j \) is a uniform random deviate in the range, \((0,1)\). D is the dispersion factor in the range, \((0,1)\). \( V_j = \text{Vector2D}(\cos(\tau), \sin(\tau)) \) where \( \tau = (\theta + \pi/2) \). Particles do not interact directly. Yet the system generates new particles at a fixed rate within the envelope, \( \Theta \), and in this sense, particles are statistically correlated or weakly interacting.

The *ParticleSystem* class contains an array of *Particle* objects. *ParticleSystem::Update* invokes *Particle::Simulate* to update each particle's state. *ParticleSystem::Render* in turn delegates rendering to *Particle::Render*. The ensemble of mass particle effects produces beautiful and stylized macro-effects simulating fluids, fires, swarms, galaxies, etc.

**GEDI EXTENSIONS**

Aside from fixing bugs, we also extended the Mirus base as direct result of play testing it. Some of these extensions were initially built into student and faculty games, not Gedi proper. The enhancements were subsequently integrated into Gedi as extensions. The sections below review these extensions.

**Dynamic ABO Class**

The ABO class uses a descriptor file to define an animation. Unfortunately, the base descriptor contains a script of unlabeled parameters, which we found error-prone. Furthermore, the ABO image is immutable after it has been loaded through the descriptor file. This requires a game designer to create all possible animations in advance, which in some situations is impractical. To remedy these problems, we developed the *DynABO* class. It extends ABO and uses a descriptor file of mnemonic key-value pairs. *DynABO* also has methods to change the image dynamically.

**SAM Class and the Mappy Interface**

The Mirus base does not support scrollable game worlds. Without this feature, the entire game world has to fit in a single screen. This is sometimes impractical and furthermore, undesirable since it limits the types of games one can develop. Scrollable
games, on the other hand, are games in which the virtual world is arbitrarily larger than the physical screen. The "view port" is the visible part of the game world on the screen. The scrollable world is usually laid out in reusable rectangular images or "tiles" that are assigned to physical screen blocks. In fact, a tile map is the function, $M : T_{ij} \rightarrow S_{x,y}$, where $T_{ij}$ is a tile number or image index and $S_{x,y}$ is a screen block or fixed region on the screen.

We developed a scrollable animated map or SAM class to implement this function. This class is a container of TileABO objects that are derived from the DynABO class. In other words, the tiles are animations. To construct a SAM object, the game needs to provide the map layout including the map size, tile size, an image from which to cut the tiles, and the map function, namely, $M$. SAM handles scrolling, animations, collision detection, and rendering.

The figure below shows a screen of 2 blocks and a racetrack map, $m_kSAMRacetrack$. This map has $3 \times 8 = 24$ tiles, although there are only 14 unique tiles. In other words, maps often lead to significant memory savings since fewer tiles may be needed to build up large and varied game worlds.

The racetrack map used to construct $m_kSAMRacetrack$ resembles the following:

```cpp
Int32 aRacetrackMap[3][8] = {
    {1, 2, 3, 3, 3, 3, 4, 5},
    {6, 7, 8, 8, 8, 8, 8, 10},
    {11, 12, 3, 3, 3, 3, 13, 14}
};
```

The snippet below scrolls the game world by one pixel left and down and renders the resulting view port:

```cpp
m_kSAMRacetrack.Scroll(-1, 1):
m_kSAMRacetrack.Render();
```

Creating maps by hand, even if they are small like the one above, is tedious. Level editors automate map construction. A popular open source level editor is Mappy. [11] It allows level designers to create and edit game worlds in a visually oriented manner. In fact, the goal of Mappy is to generate maps like aRacetrackMap. Yet Mappy can do much more. For instance, Mappy can create multiple map layers with animated tiles, collision properties, and other features. We designed the SAM class to interface directly to Mappy. Mappy exports the map and other properties as a .cpp file. This file is compiled with the game code and SAM interprets the C-structures at run-time.
Text Class and Bitmap Font Builder Interface

The Mirus base does not support writing text to the screen. We developed a class, Text, for this purpose. The `SetText` method takes a string argument. The `SetPosition` method positions the Text object on the screen and `Render` displays the string in the selected font. The class comes with three, pre-installed fonts: Small (default), Medium, and Large.

The Text class has a method, `InstallFont` that takes a FontABO object as the parameter. FontABO extends DynABO. It allows new fonts to be installed at run-time. FontABO interfaces to the font files generated by Bitmap Font Builder. [3] Thus, it is possible to easily create fonts of different sizes, colors, and styles. The only constraint is that fonts must be constant width.

Other Extensions

We created a number of other extensions. Some are listed below:

- **Random.** This class generates random deviates. It differs from `rand`, the C standard library function, in that Random generates both uniform and Gaussian deviates. Furthermore, it has no global side effects, that is, each Random object is seeded independently for independent game functions.

- **{Keyboard,Mouse,Joystick}State.** These classes, which extend the respective base classes, provide a FSA for accessing device keys that are pressed and released.

- **SoundBuffer.** This class extends the Sound class and provides methods to access the playtime, the number of channels, etc.

- **MediaPlayer.** This class can play 23 different sound formats like MP3, MIDI, WMA, etc.

- **Exception.** This is a hierarchy of classes to provide structured ways to handle errors.

- **AlarmClock.** This class provides a structured timing mechanism based on the listener (i.e., callback) pattern.

- **ReusableEntity.** This class extends Entity. Its can be reused, that is, its velocity, gravity, friction, etc., can be fully reset to initial conditions.

We also documented all classes in Gedi using the tool, Doxygen [7]. Doxygen is similar to Sun's javadoc [14] in that it produces hyperlinked class specifications.

APPROACHES AND EXPERIENCES

We believe there are at least two ways to use Gedi for teaching purposes. We discuss them below.

Game Development Platform

The Computer Science major at Marist requires students to take a course in object-oriented programming in C++. This course is also a prerequisite for GDP I. Still, we started GDP I by reviewing C++ and the mechanics of building and debugging
applications. In parallel with these activities, students began analyzing and documenting a 2D game design of their choice from a list of options. The rest of GDP I, approximately two thirds of the course, was devoted to game programming using a minimal or core set of Gedi. The core includes extending the Window class, setting up the game loop, creating animations, manipulating ABO objects, and applying linear forces to Entity objects.

By the end of GDP I, students had mastered these aspects of Gedi. Students built prototype games that implement play rules, real-time control, animations, collision detection, etc. per the design reference. However, these proto-games did not typically have sign-on screens, start-up menus, sounds, or scoring — features of complete games.

For GDP II, we gave students options to start a new game project or continue the game project from GDP I. We also studied the remaining features of Gedi such as installing and using fonts, tile maps, level editing, rigid body physics, and particle systems. We used the second-half of each class as a "game lab" in which we worked at the PC, reviewing and debugging code and discussing implementation issues.

By the end of GDP II, every student that continued his or her GDP I game project produced a complete game. Some projects used only the core features of Gedi. Others made extensive use of tile maps and physics.

**Game Engine Design Case Study**

In GDP I, we learned that students could use Gedi effectively without knowing how it works internally. Yet, in our opinion the simplicity and compact code base of Gedi are compelling reasons, indeed, invitations, to explore its internals. Such a study might be taken up in the interests of extending Gedi, designing an entirely new game engine, or providing a basis for comparison with other game engines.

One could also pursue this study for its own sake, that is, to better understand Gedi. In fact, some students encountered problems from using Gedi incorrectly. A typical example involved creating ABO objects that are not $2^n \times 2^n$ pixels in size for $n=1,2,...k$. This is a seemingly arbitrary restriction, that is, until one learns that internally Gedi implements the ABO class using Direct3D® textures. Textures support handy ABO methods like resizing, rotation, and transparency.

Based on these experiences, we decided to cover Gedi internals in GDP II. Thus, we studied the Graphics component, color theory, color keys, and textures that support the ABO class. We also studied the Input component, that is, Direct3D® devices that support the Keyboard, Mouse, and Joystick classes. Finally, we looked at the mathematical programming underlying the Physics and Math components.

These excursions, which limited time afforded us, proved worthwhile. Some students felt not only encouraged to experiment with Gedi features, creating surprisingly rich effects in their projects. Some students seemed to take ownership of and responsibility for Gedi by researching design limitations and fixing residual, internal bugs.
CONCLUSIONS

Gedi served its purposes for teaching basic game design and programming principles using a modern approach. Student contributions have been incorporated into Gedi in the forms of fixes and extensions as well as demonstration games. The Gedi project is available at http://gedi.sourceforge.net.

REFERENCES

4. Bourge, D.C., Seeman, 2004. G., AI for Game Developers, O'Reilly

ACKNOWLEDGEMENTS

We are indebted to Teixeira de Sousa for creating the original Mirus and to the students for advancing Gedi through the game project in GDP I and II.