Reducing Video Game Creation Effort with Eberos GML2D

Frank E. Hernandez and Francisco R. Ortega

The effort of developing a game has increased in the past 30 years from a task that could almost be entirely handled by a single programmer to an endeavor requiring a large team. This increase in effort has led to an increase in the cost, to the point that some have gone from amortizing the cost among multiple platforms to amortizing it across multiple future sequels [Dickheiser 06]. Now, more than ever, is time to explore, and produce tools that aim at reducing this increasing effort.

This article presents Eberos Game Modeling Language 2D (Eberos GML2D), a graphical domain-specific language for the 2D game domain. Our article outlines the constructs necessary to develop such a language for the domain of 2D games, as were found through our development of the Eberos Game Modeling Language 2D. By using this intermediate language for modeling our games, we have found a reduction in not only the amount of code required for writing a game, but also in the amount of work required to develop the same project in multiple platforms. We also discuss how, by using such a language, we have been able to reduce our game development effort by as much as 86.4%.

1.1 Model-Driven Engineering

The term Model-Driven Engineering (MDE) refers to software development appraches in which abstract models of software systems are created and then transformed into implementation [France and Rumpe 07]. In MDE, domain-specific languages (DSL) are languages developed for a specific problem domain. Unlike general-purpose languuages such as C++ and Java, a domain-specific languages can only solve problems inside the domain for which they were developed. In this article, we present a DSL for the domain of 2D games built to abstract the main concepts of this. This

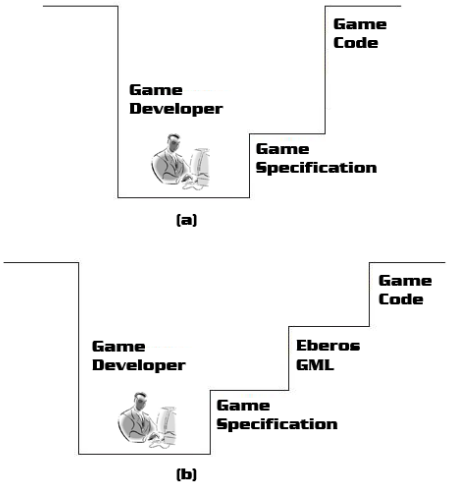

Figure 1.1. Gap between game specification and development. (a) Current gap, where developers must rely on a combination of intuition and experience when bridging the gap between the stages. (b) Eberos GML aims at reducing the gap by providing developers with an intermediate stepping stone between stages.

language can then be automaticaly translated into implementation source code, largely reducing the effort in developing said 2D games.

1.2 Eberos Game Modeling Language 2D

In this section, we discuss the concepts we found to be common for all 2D games during the development of the Eberos Game Modeling Language 2D. It is important to note that this is by no means the only or best language approach to modeling 2D games, but rather is the approach we developed. This approach is also by no means complete; we present it in this article with the hope of demonstrating the savings in development that similar approaches can give your project.

While exploring we have identified the following requirements for the game modeling language:

Simplicity: Part of the reason in the increase in effort of developing games is the overhead for translating the game requirements into source code [Fig. 1.1a]. Eberos GML2D aims at reducing this cost by providing an intermediate step between game specification and game code [Fig. 1.1b].

Platform-Independent: Since the game logic is independent of the underlying platform, so too must be the language that expresses it. Eberos GML2D does not model any platform-specific concepts of the underlying platform layer.

Library/Game Engine-Independent: Similarly, the modeling language must not be linked to any specific development library or game engine.

Expressiveness: It must be able to model a large majority of game development projects.

Before we move on to explain the constructs of Eberos GML2D, let's first look at the specification for a simple game of Pong for the PC. First, we'll show how it would look in plain English, and then how it would look with Eberos GML2D.

English: This game of Pong is composed of two players. Each player controls a paddle at each side of the screen. Player1 controls the paddle at the left of the screen, while Player2 controls the paddle at the right side of the screen. Player1's controls are the 'A Key' for moving the paddle up and the 'Z Key' for moving the paddle down. Player2's controls are the 'UP Key' for moving up and the 'DOWN Key' for moving the paddle down. The ball in the game will start at the center, and will begin each round with a random direction. When the ball crosses the left or right limit of the screen, the opposing player will gain a point and the ball will reset at the center. The game will be a windowed game with dimensions of 800 pixels wide by 600 pixels high.

Eberos GML2D: The game has two ***UserDefinedEntities****; 'Player' and 'PongBall'. The 'Player' entity has one 'Sprite' with the bar graphics. It has a global state which listens to the 'MOVE_UP' and 'MOVE_DOWN'* ***GameMessages****. The 'Player' entity has a* ***BoudingRectangle*** *for collision detection. Finally, it has two 'CompositeActuators'; 'MoveUp' and 'MoveDown', which handle the translation of the entity up or down. The 'PongBall' entity has one* ***Sprite2D*** *with the ball graphic. It has a* ***BoundingRectangle*** *for detecting the collision with the player's paddle and the screen limits. The 'PongBall' entity has three* ***CompositeActuators****; 'MoveBall', 'ResetBall', 'RandomizeDirection'. The 'PongBall' entity also has two* ***States*** *'BALL_RESET' and 'BALL_ACTIVE' for supporting the logic of the 'PongBall' entity. Since the game is a windowed game, the GameRoot entity is 800 pixels wide by 600 high. The GameRoot also has three* ***EntityReferences*** *'Player1', 'Player2', and 'PongBall'.*

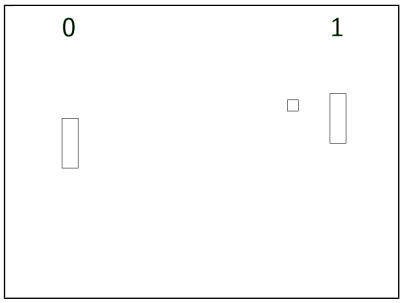

Figure 1.2. Simple Game of Pong.

*The player references are then controlled by two **InputHandlers**, 'Player1Input' and 'Player2Input'. The 'Player1Input' responds to the 'A' and 'Z' keys, and notifies the 'Player1' reference of 'MOVE_UP' and 'MOVE_DOWN' **GameMessages**. Similarly, 'Player2Input' responds to the 'UP' and 'DOWN' keys, and notifies 'Player2' reference of 'MOVE_UP' and 'MOVE_DOWN' **GameMessages**.*

Source Code: Once the game has been defined, the Eberos GML2D model created can be automatically translated into source code, and the initial game version is created [Fig. 1.2]. The game might still require some programming to complete, but you have just saved yourself the repetitive work of creating the states, implementing the input handling logic and collision detection. We discuss these benefits in the last section of this article.

1.2.1 Entities

Entities in Eberos GML2D are akin to actors in a game. The same way that actors are the meat of all games, entities are the meat of Eberos GML2D. Entities are used to model the actors in a game. There are two kinds of entities in Eberos GML2D, user-defined entities, and specific entities.

GameRoot
The GameRoot represents the entry into the game; this is similar to the main function in a C++ program. In Eberos GML2D, the game model is viewed as a graph, where at least one of the entity edges must be connected to the GameRoot. The GameRoot is also used to specify an initial set up about the game. Currently, the GameRoot entity specifies the screen resolution, and whether the game is windowed or full screen.

GameMessageTracker
The GameMessageTracker entity controls all the possible GameMessages

that exist in the model. This entity ensures that any message by any of the other entities is a valid message. Game messages in Eberos GML2D are represented by the GameMessage entity. Just like in any game, messages in our language represent a single, specific piece of data.

GameMusicManager

Similar to the GameMessageTracker, the GameMusicManager entity specifies the music and sound effects available in the game model. The GameMusicManager also control which sounds and music can be played by the PredefinedActurators. This entity is populated by SFXTrack and BGMTrack entities, which specify information specific to the audio file. The 'Track ID' is unique indentifier that allows for each specific track to be used by the 'PredefinedActuators', while the 'Track URL' is used to specify the location of the music and sound effects in the system. The 'Track URL' is used when translating the Eberos GML2D model into source code to transfer the file over to the project's folder. This is done as a means for reducing the errors of specifying the project's path to each sound file.

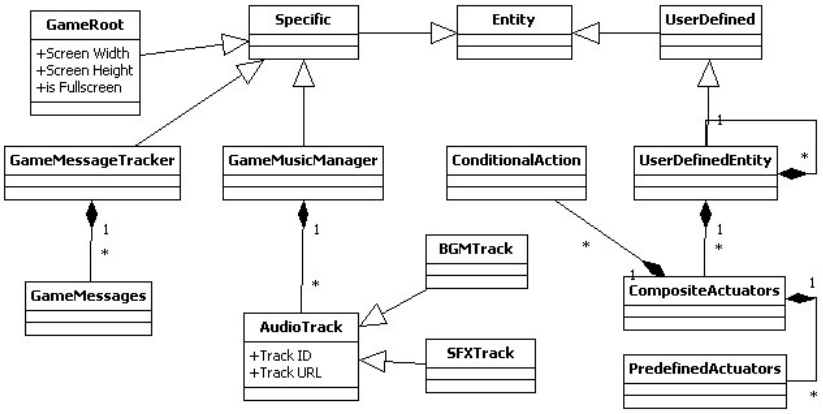

Figure 1.3. Basic class diagram with an overview of the entity relationships.

UserDefinedEntity

UserDefinedEntities [Fig. 1.3] are meant as the catch-all of entities, and can be used to model any of the specific entities. Entities are the center of Eberos GML2D; everything is treated as an entity in the game model. Since entities/actors can range from menus, to dialog choices, to players, and much more, entities must support a range of properties common to all of these actors. UserDefinedEntities can be composed of zero or more entities, which allows for the modeling of nested entities, such would be the

case of a menu composed of menu choices.

Entities are also composed of CompositeActuators. CompositeActuators represent actions that the entity can perform in the game. CompositeActuators are composed of PredifinedActuators, which are a small subset we have found to be common to all 2D games.

Below we list the PredefinedActuators found during the development of Eberos GML2D:
Translate: Displace an entity by a given amount.
Place At: Place an entity at the given location.
Notify Self: Allows the entity to send itself a message.
Play/Stop Sound: Play/Stop a sound effect.
Play/Stop Music: Play/Stop a music track.
Invert Axis: Invert a given coordinate axis.
Play/Stop Animation: Play/Stop a given animation.

Actuators can also have ConditionalActions, which represent actions that only occur if a specified condition is met. These are composed of two parts: conditions and actuators. The condition evaluates to true/false, and the actuators are a set of Predefined/CompositeActuators to be executed when this condition become valid.

Below is a list of conditions found during the development of Eberos GML2D:
Check Collision: True if a collision has happened between the entities.
Check Position: Check if the entity position is at or around the specified position.

Finally, entities can also have: zero or one **State/GlobalState** representing the initial state of the entity's state machine (Section 1.2.2), zero or one **Sprite2D** (Section 1.2.3), zero or one **BoundingObject** for collision detection(Section 1.2.4), and zero or one **InputHandler** for modeling input.

1.2.2 Logic

The game logic is probably one of the most important and most time-consuming aspects of any game project. While there are many approaches to implementing the behavior of actors in a game, finite state machines (FSM) seem to be one of the more versatile approaches available. In Eberos GML2D, we have two constructs, **State** and **GlobalState**. These constructs were modeled based on the state architecture presented by Mat Buckland in [Buckland 05]. Similar to his approach, States and GlobalStates have one owner entity [Fig. 1.4].

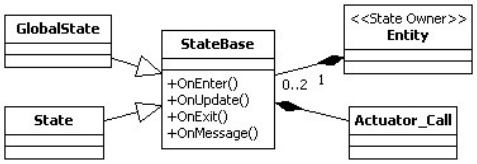

Figure 1.4. Basic class diagram for entity to state relationship.

State/GlobalState

A State/GlobalState directly attached to an entity in the model means that such a state is the initial state for that FSM. Each state is composed of Actuator_Calls, which represents actions that entities can take in the game. This actuators can be placed inside the OnEnter, OnExit, OnUpdate and OnMessage sections of the states. As the names implies, Actuator_Calls placed inside the OnEnter section execute when the state is entered. Any Actuator_Call placed inside the OnUpdate section is called on every update, and any Actuator_Call placed inside the OnExit section is called when the state is being exited. The OnMessage section of the states is a bit different from the other sections. Any Actuator_Call placed inside of this section must have a GameMessage specified along with it, and is only called when that GameMessage is received by the state.

A finite state machine (FSM) is built in Eberos GML2D by linking states together with TransitionEvent links. Currently, a TransitionEvent link represents an edges from **State** to another **State**, or a **GlobalState** to another **GlobalState**, and specifies the message that triggers the transition from such state to another. Finally, a FSM modeled in Eberos GML2D is only valid if every state in the Eberos GML2D game model is reachable.

1.2.3 Sprite and Animations

In Eberos GML2D, Sprite2Ds [Fig. 1.5] represent the resource image file for a sprite or sprite sheet. This construct holds the information specific to that file, such as the resource URL, the image width and height (in pixels), and where in the screen it will be drawn by default. The screen destatination is used specially in the case of modeling a HUD or a menu. In the case when the sprite is actually a spritesheet containing multiple animations, this can be modeled using one of the two animation constructs, **AnimationStrip2D** or **CompositeAnimation2D**.

Animation2D

Animation2D is an abstact representation of all animations, so every an-

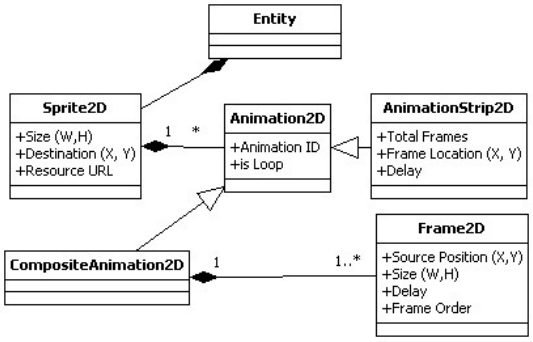

Figure 1.5. Basic class diagram for entity to sprites relationship.

imation construct in the language is derived from this. Every animation contains an animation ID for addressability purposes inside the language, and information as to whether or not the animation loops.

AnimationStrip2D

We have found that in most cases of a spritesheet, the animations tend to be on a given line from left to right. Eberos GML2D models these with the AnimationStrip2D construct. The AnimationStrip2D construct represents a single strip of frames on the spritesheet. The strip is composed of uniformly sized frames. This is a somewhat rigid construct that allows rapid specification of simple animations. Each instance of this construct contains information about a single animation such as: the total number of frames, the size of the uniform frames, the position in the sprite image where the first frame appears, and the delay (in milliseconds) of each frame.

CompositeAnimation2D

There are also the times when an animation inside the spritesheet might be made up of frames of different sizes, or frames that are not necesarily adjacent to one another; to model this kind of animation, we have the CompositeAnimation2D construct. A CompositeAnimation2D is another kind of Animation2D, and is used to represent animations at a deeper level of detail. Each CompositeAnimation2D is composed of one or more Frame2Ds.

Frame2D

A Frame2D represents a single frame of the animation. This construct gives a lower level of control over the modeling of the animations by allowing their spcification of an animation on a frame-by-frame basis. Each

Frame2D contains information about its X and Y position on the source spritesheet, its width and height, its delay (in milliseconds), and its position in relation to other frames in the animation.

1.2.4 Collisions Detection

Currently, the language only models bounding object collision detection with bounding rectangles and bounding circles. An entity in the game model becomes a candidate for collision detection when a **BoundingObject** is defined for it [Fig. 1.6]. This means that now the entity can be a target for actuators that deal with collisions, as is the case for the **ConditionalActuator** CheckCollision. For Eberos GML2D, we only model two kinds of **BoundingObjects**; **BoundingRectangle** and **BoundingCircle**. The **BoundingRectangle** contains information about its width and height in pixels. Similarly, the **BoundingCircle** contains information about the size of its radius in pixels.

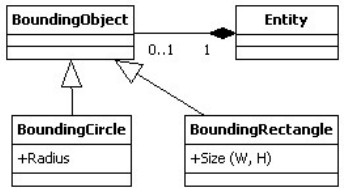

Figure 1.6. Basic class diagram for entity and collisions relationship.

1.2.5 Input Handling

In Eberos GML2D, every entity reference is capable of having an **InputHandler** construct attached to it. When an entity has an **InputHandler**, it means that such entity is controlled by an input device in some way. At the time this article was written, **InputHandlers** only model keyboard input.

Keyboard input is modeled by adding a **KeyboardKey** to the **InputHandler** [Fig. 1.7] construct. In Eberos GML2D, a **KeyboardKey** contains three pieces of information: the key to detect, the state of the key we are interested in, and the **GameMessage** to dispatch when the key and key state are matched. Currently, there are four possible states that can be modeled: **DOWN**, **RELEASED**, **WAS_PRESSED**, and **IS_HOLDING**. Once the input is matched, that is, the key specified is in

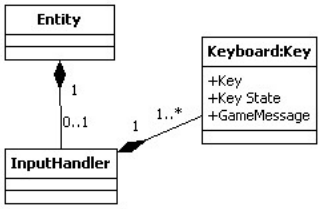

Figure 1.7. Basic class diagram for entity to input relationship.

the desired state, the entity attached to the **InputHandler** is notified of the **GameMessage** specified for that key. This **GameMessage** is then used as a link between the entity logic and the input.

1.3 Reducing Effort

Up to this point, we have presented the concepts we abstracted during the development of Eberos GML2D, which we found common in the 2D game domain. By doing this, we have taken would-be repetitive work and encapsulated it into simple language constructs. In implementation terms, we have taken what would have been repetitive code, and reduced it to a simple operation. Also, by developing a language, we are now constrained by the language semantics, and have reduced the number of errors that could made during the development of said repetitive tasks. However, before we can profit from our language, we must first develop the means to translate it into implementation artifacts, such as source code or any other artifact specific to the desired technology [Fig. 1.8].

The final step before we can enjoy our effort savings is to develop a translator for the language. This last step, however, does require some platform knowledge by the translators. That is, the translator must know what is the equivalent of the language on the target platform or engine. In our specific case, we have implemented translators for Microsoft's XNA and SDL, so our language constructs translated to either a set of classes, or a set of function calls specific to the libraries.

In the case of our implementation, this abstraction meant reduction in the amount of code that was needed for items like state machine, animations, and game logic [Fig. 1.9], and an overall reduction of as much as 86.4% in the entire game code. Before Eberos GML2D, programming a state machine meant we would copy and paste most of the architecture code for each state class, and then modify it to add the code specific to the

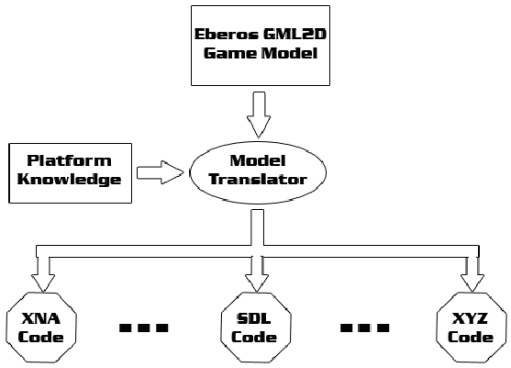

Figure 1.8. Model translation: Once the game model is created, it can be translated into source code for different underlying platforms.

logic we wanted to implement. For state machines contaning 20+ states, this was time-consuming and error-prone as mistakes would inevitably be made when linking the right states together. Currently, this process is developed graphically using Eberos GML2D, and the code is automatically generated, thus reducing the chances of programmer errors. Also, with the additions of the CompositeActuators, a large set of the code specific to each state has been reduced. Similarly, the code for input handling, animations and collision handling has beem partly automated.

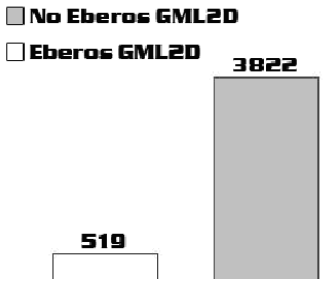

Figure 1.9. Number of lines of code a developer needed to write the same game to completion, both with the aid of Eberos GML2D and without. Results from experimental game SpaceKatz.

1.4 Conclusion

This article describes the constructs, which we found during the development of the Eberos Game Modeling Language 2D to be common among all 2D games. It also describes some of the requirements to keep in mind while creating a game modeling language, and the possible effort amount such a language could reduce. The Eberos GML2D presented here is not a complete language; there are still many concepts common to 2D games that it does not encapsulate. At its current stage, however, it has shown some promising results, and we hope that it will entice the reader to further explore the benefits that domain-specific languages can bring to the game development process.

Bibliography

[Buckland 05] Mat Buckland. *Programming Game AI by Example*, First edition. Plano, Tx: Wordware Pub, 2005.

[Dickheiser 06] Michael Dickheiser. *Game Programming Gems 6*. Hingham, Mass: Charles River Media, 2006.

[France and Rumpe 07] Robert France and Bernhard Rumpe. "Model-driven Development of Complex Software: A Research Roadmap." In *FOSE '07: 2007 Future of Software Engineering*, pp. 37–54. Washington, DC, USA: IEEE Computer Society, 2007.