GameMaker Language and Its Applications

Author: Logan Brandt

Abstract:

GameMaker Language is a programming language that was designed specifically to create 2D games. This has impacted design choices in the language and what features it should include. Because of this, GameMaker includes some functionality other languages do not have. This, however, has led to some things the language is simply not capable of.

Logan Brandt 12/2/2019
Programming Languages Project Paper

Outline:

Introduction:

- Talk about history of Gamemaker
- Find a source that isn't wikipedia saying it was originated in C++/C#
- Talk about how this language has strengths and weaknesses

Body:

GUI:(3-4 pages)

- Talk about the GUI for GML studio 2
- Discuss the different tabs that don't have depth that you would use:
    - Sprites
    - Sounds
    - Fonts
- Start going into the tabs that have a lot of depth
    - Objects
    - Room

Objects:

- Discuss what features are in objects
    - Include here the collision story.
- Compare objects to those in Java

Rooms:

- Discuss what rooms are, since they're not in other languages.
- Talk about how powerful they can be.

Unique keywords:(1-2 pages)

- Language can operate without semicolons by filling them in for you, so missing one doesn't crash the program.
- Discuss what keywords and attributes objects have that make this unique.
    - Keywords:
        - Other
    - Attributes:

        - Gravity
        - Lives
        - Score

    Weaknesses:

    - Talk about how the software is not for 3D graphics.
    - If you are doing too many calculations in the step event, it will not compute it fast enough (a very large number)

Conclusion:

- Talk about how the project was fun to make and implement
- Discuss what can be done in the future.
- Discuss what I want to do with the language in the future.

The language that I have chosen to work on for this project is GML, also known as GameMaker Language. GML was developed by Yoyo games, and the first version of it was released under the name Armino. Since then, the language has grown a lot and been updated multiple times. Currently, the environment for the language is called Gamemaker Studio 2, and uses a Graphical User Interface to help the users. In this project, I explored different features of GML, and how it has implemented these features to help make the task of creating a game easier for developers. I also explored what GML does not do, as it is not to be used for developing every type of game.

The first important thing to consider about GML is what it was designed for: making games. This design goal impacts everything about the language, such as what variables to use, what methods to include, etc. Because of this, there are methods and variables present in this language that are not in others. For example, there is the "lives" variable, which can be used to store the lives of the player. This variable is innately global, as in it can be used anywhere in the program immediately, but it does not have to be specified by the programmer. GML is one of the only languages I know of that automatically creates global variables for the programmer to use. This is an example of them creating their architecture with their intended audience in mind.

The next thing to consider about GML is it's layout. For GML, there is a Graphical User Interface with many different tabs and windows for different features of the language. This is vastly different from other languages, as they are not nearly as reliant on GUI's to navigate the different areas of the code. In GML, there are a list of different pieces to code called resources. As of Version 2.2.4.374, there are 15 different resources. Each resource is for a different aspect of making the game, and each can do different things. Some of these resources are simple. One example is the Sprite resource. This allows the user to store sprites, which are 2D drawing that could depict anything shown on screen, from the character to the environment to weapons used by the character. This tool has an option to let you create your own sprites using a rudimentary tile system. It also allows for multiple frames of the sprite, or different positions the sprite can have. This can allow for sprites to change their position to look animated or to change how they look based on the action they perform. There are many different actions that can be taken with just the sprite resource, and it is not even the most complex. The two most complex, and arguably the most important, are object and room.

The object resource stores an object in the system that the creator can influence and manipulate. This resource holds many powerful tools, and is the main power behind creating game logic and making things happen. When you first create an object, you can assign it a sprite, which will be what the object looks like when it is first created. Inside of the object class, there are a plethora of events that can occur. These events are things that occur upon specific timings or when they are called. For example, there is a create event that will execute whenever the object is first created. This functions like a constructor in other languages such as Java, and allows for declaration of local variables that can be used elsewhere in the programmer. A more unique event that the object resource has is the Step event. This event is important for allowing for changes to happen in the game in real time. The way it works is that it occurs for every frame of the game. Frames operate by showing one image of a game, then another, and another, while adjusting what is happening on the screen between each frame based on player input and code for the environment. For example, if a game operates in 60 frames per second, then the step event would be called 60 times, and it's code would run 60 times. This is where user input would impact an object to move, for example, or for an AI to use existing code to move as well. While different objects can have different code in their step events, step will occur the same number of times for both of them, meaning that the objects will "step" at the same time. These are just two different features of the object resource, but there are many more that exist that can have a major impact on the object they are coded in.

The object resource has similarities to other OOP. Since the basis for the code was in C++, GML was able to use the code to make objects in their own way. Each object has its own set of variables, and these variables can be made independent of the object. Many of the things that can occur with objects in other languages can happen here, including things such as inheritance. Another important thing to note is that not all objects have to have a sprite, as there can be invisible objects that work in the background while the game occurs in the foreground. Invisible objects are paramount for adding logic into the game in ways that objects the players sees could not accomplish. A common feature for invisible objects is for them to keep track of the score to see if a player has won or run out of lives. They can also set up alarms. Alarms are events that are triggered when an alarm is set somewhere else, and then goes off after a certain amount of time. This is a way for different objects to directly interact with each other, as an alarm in one object can activate the code in a different object. GameMaker has 12 alarms that can be programmed in, but alarms are not unique code. An alarm going off can trigger multiple objects at once if they have code for it, which can make it pretty useful. An example to use this is if a player creates an explosion that destroys everything on the screen, having an alarm could allow for everything to get destroyed all at once. This solution seems far easier than something that could occur with objects in a language such as Vizard. For Vizard, you would have to loop through and delete all the objects manually if the event were triggered. Objects can interact with other objects and do some crazy things, but only under one condition: they have to be in a room.

The room resource is where all of the gameplay occurs. In order for an object to be able to execute, it must be added to a room. Objects not in the room will not execute, as they will not be created, and thus won't be able to run. To add an object to a room, a creator can drag it from the right side where it is shown and drop it into a tile. It does not matter where invisible objects go, and they will not impair the movement of other objects on screen. This can also be done with sprites, but sprites will not move once placed into the room.

There are a number of unique features in GameMaker with the idea of game design in mind. Some of these include things such as specific keywords in the language, such as with or other. Some alternative features can be oriented towards making things easier for the programmer, such as some built in functions. Overall, there are a number of languages that are designed to make the language easy to use, yet with enough tools to allow for more complex features.

This language has built in functionality to test if two objects have collided. Provided that both objects have a collision mask on, if these collision masks overlap, they can trigger an event. While making changes to the object the collision event in is easy, it can be difficult to figure out which object it is colliding with. Referring to the object instance will refer to all objects of that type, which is not ideal. To solve this issue, this function utilizes an extremely useful keyword: other. This refers to the exact instance triggering the collision, allowing changes to be made to a single instance of an object instead of all of them. This saves a lot of time for the programmer since they don't have to make calculations for both sides of the collision to figure out which object is run into.

This reminds me of programming in Vizard, which uses Python as a basis for its code. This was in Computer Graphics, and we had to program a collision between a ship and a bullet. This was a large task, which involved loops to go through each alien ship and manually compare the (x,y) coordinates of each bullet created with a ship to find a collision. Finally, we could write code to delete both objects from a list for both the bullet and ship. In GML, it would have been only 4 lines to destroy both the bullet and the ship it ran into, with no comparisons on my end. This is an example of GML being better designed to handle game logic like this than other languages. While Vizard's more in depth coordinate system allows it to run in 3D better, GML's built-in logic handles operations like collision smoothly.