A Component-based Game Engine for the Game-O-Mat Game Generation System

Mike Treanor
American University
treanor@american.edu

Abstract

This paper describes the game engine of the forthcoming game generation system titled *The Game-O-Mat*. As underlying game engines strongly constrain the space of possible games that a generator can create, the goal of this paper is to fully document the design considerations and features of this game engine that was designed with game generation as a priority. This paper is aimed at helping future developers of game generation systems.

Introduction

The purpose of this paper is to document the features and design considerations that have gone into the forthcoming game generation system titled *The Game-O-Mat*. *The Game-O-Mat* system (which is heavily inspired by *Game-O-Matic*) (Treanor et al. 2012), has been designed with the goal of enabling the easy creation of games that represent ideas through gameplay metaphors built from graphical logics (Wardrip-Fruin 2020). Rather than evaluate the system's generator (which has been in concurrent development), this paper will compare and evaluate the design of the game engine itself. In the game AI research field, a spects s uch a generator's game engine are often only loosely described, and regarded as implementation details, as they do not directly describe the use of particular artificial intelligence techniques. While it is true that the design of a game engine is arguably a matter of software engineering and game design, it is a mistake to not recognize that the underlying engine of a generation system strictly constrains what kinds of games a system can generate, as the generator needs to manipulate the engine to create games. Furthermore, the desirable features of a game engine designed for generators to manipulate will be different than the desirable features of a game engine designed for human creators.

A large part of the design and development of this component-based engine involved determining the set of behavior components, and how they would interface together. Ivey et al. encountered a similar scenario while developing their physics-based parameter driven game system, *Gamika* (Ivey et al. 2018). Those researchers shared many of the same goals of this project and they presented a methodology for designing engines targeted for game generation. Ivey et al. describes the process of "capturing inspiring examples" by adding many parameters freely and then refining them through consolidation. *The Game-O-Mat*'s engine was likewise augmented and enhanced by implementing several classic, yet simple, games, and features and components were added as needed. This process was applied until it became unlikely that a generator would be able to create the game examples. This paper concludes with an advanced example that shows the limits of what the generator might be able to handle.

Another project very close to *The Game-O-Mat* in nature is *Gemini* (Summerville et al. 2018), an impressive generator that made use of computational *interpretations* of its games as it generates. During development, the engine itself was given to human developers who were tasked with creating games. As described below, this process was lightly emulated as sample games of increasing complexity were hand created to test the engine. Other game generation systems and game description languages exist and provide valuable insights. For example, some incarnations of ANGELINA (Cook, Colton, and Gow 2017) were designed with genre-specific goals (e.g. Metroidvania), and the VGDL project was designed around creating games that could be played and evolved using automated players. Similarly, Duplantis et al. similarly present an example of a genre specific game description language for role playing games in the style of the games made for Game Boy (Duplantis et al.).

Game generation is arguably one of the most difficult challenges for procedural content generation because there are so many interrelated facets involved (Liapis et al. 2019). There will be no solution to this challenge, and instead researchers will present a set of generators that support different aesthetic experiences of varying levels of complexity. This paper simply hopes to fully document what the games of *The Game-O-Mat* are made out of in order to help future creators of game generation systems.

Engine Features

At the core of *The Game-O-Mat*, and most component-based game engines, are game objects and behavior components. Game objects are typically the objects that move around on the screen that collide with, and create other game objects. Behavior components control what these game objects do and how they interact. Components are *modular* in that they are created to operate independently. The idea of viewing certain games as graphically represented entities that behave according to rules, and that react when they collide/overlap with one another is central to what Wardip-Fruin describes as games with graphical operational logics (Wardrip-Fruin 2020). The game engine for *The Game-O-Mat* is designed to be able to represent the space of simple one-screen 2d action games comprised of graphical logics (such as those that were played on the Atari 2600). In addition to components, game objects can be associated with variables that either can be used as blackboards for inter-component communication, or a mechanism to track an individual game object's state. Game object variables can also be used and displayed as part of the game's mechanics (see Orbital example below).

To add concrete ways for games to end, this engine also provides mechanisms for a game to terminate (i.e. be won or lost). These win-and-lose conditions make reference to either a counter or a meter. Counters are numerical values tracked by the engine, and conditions are comprised of simple equality/inequality checks on their values. Meters are visually represented counters.

Components

The set of components, their ability to adapt, and their expressivity define the range of possible games that can be developed in a component-based game engine with predefined components (as is the case with *The Game-O-Mat*). *The Game-O-Mat*'s engine particularly strives to provide a minimal set of components that are adaptable to many circumstances without requiring an inordinate amount of configuration. The set of components of the current *The Game-O-Mat* engine was arrived at through balancing the conflicting priorities of generality and ease of configuration.

As a concrete example of these conflicting priorities, consider a single "Tank Controller" behavior component that would make any game object rotate when the player pressed left and right, and move forward and backwards when the player pressed up and down. This very common player controller behavior could also be represented by two different components that simply rotated on input, and a second that moved forward on input. *The Game-O-Mat*'s engine chose to avoid components of the "Tank Controller" variety, and instead abstracted the conditions under which a component would be *active* to allow for components to focus on one task at a time (described below). The point of highlighting this choice is to emphasize this unavoidable engine design challenge, and to motivate the design of the activation system described below.

This design decision is a departure from *Game-O-Matic*. *Game-O-Matic* made use of less general components and thus was able to make more use of descriptive component tags. For example, if there were to be a "Tank Controller" component, it could be tagged as a component that enables movement, and the generator could know that there was a game object to control in the game when it saw that any game object had a component with that tag. *The Game-O-Mat*'s generation engine will need determine that there is a controllable game object using a more sophisticated method.

Component Parameters Each component has a small set of parameters that configure the component's behavior. Examples might include the "Rotate" component's rotation speed, or the amplitude of the "Movement" component's y axis sinusoidal movement. Each parameter is assigned a default value, and for numerical parameters minimum and maximum values. Furthermore, the engine is able to translate terms such as "fast", "moderate", "very fast", etc. into appropriate numerical values for parameters. Additionally, a range can be provided that will be randomly resolved when evaluated. These details are very important, as the generator will struggle to make fine adjustments, and the ability to coarsely and minimally specify games is essential for the generator to operate.

Component Targets Some components control a game object with regard to some other game object. For example, the "Move Towards" component needs to know what it is moving a game object towards. This engine refers to these game objects as "targets," and a component can target an arbitrary number of other game objects, or a group of "tagged" game objects (game objects can belong to an arbitrary number of groups by sharing a tag). When a component targets more than one game object, it will default to targeting the closest one. Components can also target the mouse pointer.

Component Activators By default, when a game object is assigned a component, that component will perform its behavior upon that game object. However, components can be configured to only perform their behavior under certain conditions. The mechanism that controls these conditions, called "activators," are shared by all components and they drastically enhance a component's versatility. Once an activator's conditions are met, it will make it so the component will apply its behavior to the game object for a specified duration. If no activator's conditions are met after that duration, the component's behavior will no longer be applied to the game object. Below is an overview of the currently available activators:

- **Collision:** A component is only active if it is graphically overlapping with a targeted game object.
- **Distance:** A component is only active if it is within a specified distance of a target.
- **Input:** Make a component active when specified mouse or keyboard input is received.
- **Timer:** A component is activated after a specified amount of time passes. This activator can optionally reset itself.
- **Meters/Counters State:** Activate a component when a counter or meter reaches certain states (e.g. score ¿ 0).

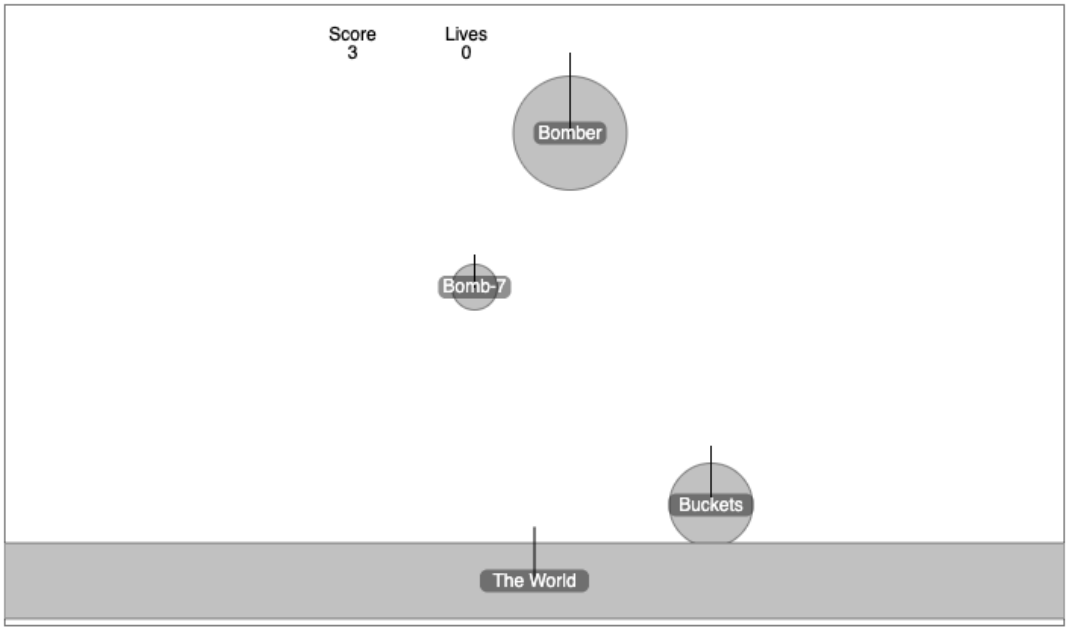

Figure 1: Kaboom! is relatively easy to represent using *The Game-O-Mat*'s game engine.

- **Component or Game Object Parameter:** Activate a parameter when one of the game object or component's parameters reach a specified state (e.g. when the game object's scale is greater than 2).
- **Conjunction:** This activator is configured with a list of activators (such as those described above). The component will only be active if all of the activators in the list's conditions are met. Note that the original list of activators is by default a disjunction of the supplied activators.
- **SRE:** Make a component active when a Scored Rule Engine (SRE) rule evaluates to true (details below).

Component Effects Effects are external changes that a component can make to a game when it is activated. The primary use of effects is to modify meters and counters, however they can also be used to spawn other game objects.

Development Platform

One of the primary goals of game generation systems described in this paper are to enable non-experts to rapidly create and share games about anything. While *Game-O-Matic* partially delivered on this goal, the platform it was developed for, Flash, is no longer supported on contemporary browsers, and thus games cannot be created or shared any longer. In response to this, *The Game-O-Mat* is being developed using ECMAScript 2015 (ES6) without taking advantage of libraries and development platforms that may become defunct. The graphics engine is currently p5, though the interface to the graphics engine is clean and *The Game-O-Mat* has also used PhaserJS. This paper is aimed at helping future developers of game generation systems, and choosing a long lasting platform to develop for, in this case the web browser without plugins, is strongly recommended.

Scored Rule Engine

Built into *The Game-O-Mat* is a rule system called the Scored Rule Engine (SRE). SRE is able to answer a wide range of questions about the game state by evaluating rules and returning truth values and bindings. While primarily used by the generator not described in this paper, these bindings can be useful for providing targets and also can modify the game state is certain real-time scenarios. The set of queries the system can make currently are:

- **HasComponent:** Does an game object have a specified component assigned to it?
- **HasTag:** Does an game object have a specified tag assigned to it?
- **Colliding:** Are two game objects currently colliding?
- **GameObjectPropertyCheck:** Does a game object have a property that meets some condition?

For an example of a SRE rule being used as an activator, see the *Orbital* example below.

While SRE and the activator system duplicate some functionality, SRE is able to make much more sophisticated queries of the game state. However, because rule evaluation requires exploring the search space of logical binding possibilities, it is computationally expensive and should be used sparingly for real-time applications (at least until performance enhancements are made). Finally, it should be noted that the engine does not make full use of the rule system (particularly the 'Scored' part which is used by the generator while making design decisions).

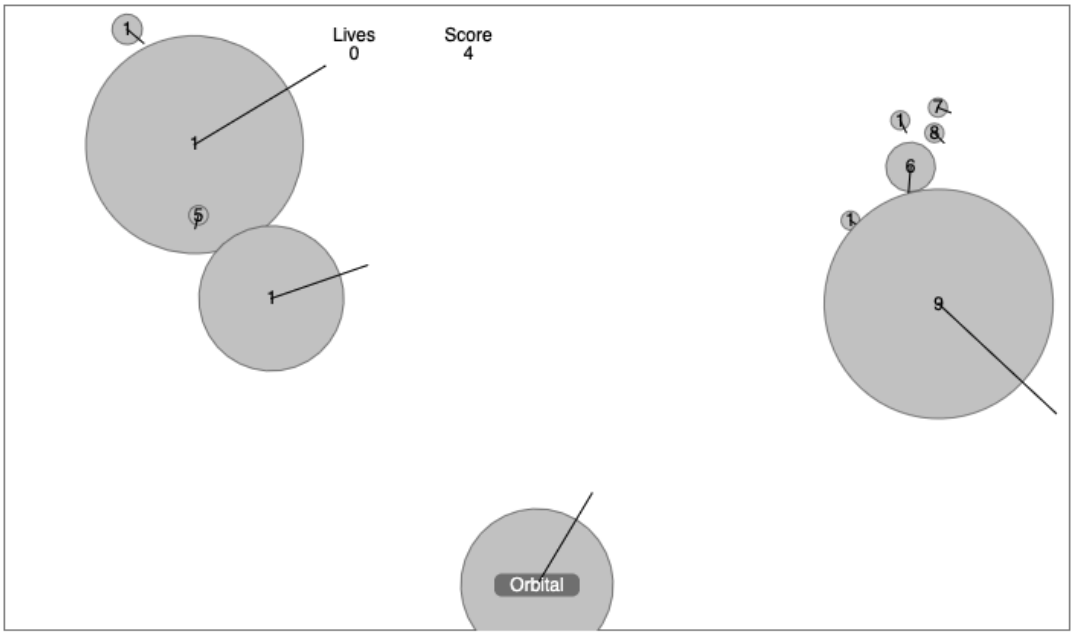

Figure 2: The iOS game *Orbital* was able to be created in this engine, but required nuanced engine manipulation that would be hard to a generator to perform.

Example: Kaboom!

Below is a description of how this engine implements the classic game *Kaboom!* (1983). In *Kaboom!*, the player controls the left and right movement of a graphical depiction of *buckets* of water using the arrow keys. The goal of the game is to catch the *bombs* that a *mad bomber* is hurling down at the *world* below. When the player (*buckets*) collides with a *bomb*, the score is increased, and when a *bomb* gets by the *buckets* they lose a life (presumably because the bomb detonated in the unseen *world* below).

The Game-O-Mat's engine represents *Kaboom!* in 210 lines of expanded JSON that makes use of basic engine features with lightly configured components. To begin with, two counters are created for *Lives* and *Score*. Next the *mad bomber* game object is created and placed at the top of the screen, and given a random x velocity (to begin it moving left or right). Next, it is assigned a "ModifyGameObjectProperty" component with parameters that will multiply the game object's x velocity by -1, as well as a timer activator that will activate the component periodically. Next, an "EdgeWrap" component is applied. These two components get the *Mad Bomber* moving left and right in unpredictable ways. Next a "CreateOnTrigger" component is applied with parameters specifying that is should create a *bomb* (details below) at its current position, and a timer activator that will control the rate that *bombs* are created.

The *bombs* game object description involves first applying a "Movement" component that constantly moves the game object downward (no activators are needed as it is desired for this to be constant behavior). Next, a "Lifetime" component is applied that will remove the game object after 6 seconds (this is done for performance reasons). Next, two "DestroyOnTrigger" components with collision activators are applied. One that targets the player that has the effect of causing the *Score* to increase, and the other targets the *world* object (described below) that has the effect of decreasing the *Lives* counter by one.

Finally, the *world* game object is a sized to be a long rectangular game object at the bottom of the screen.

This example illustrates how game objects, components, activators, and effects all work together to create a game. Other games that this engine can represent with similar complexity (as measured by lines of JSON and component/activator assignments) are *Pong* and *Breakout*.

Finding the Limits of the Engine: Orbital

While *The Game-O-Mat*'s engine can relatively easily represent games with the complexity of many Atari 2600 games, the same cannot be said for games with more sophisticated game mechanics. As an example, consider the early iOS action game, *Orbital* (2009). *Orbital* is a one-button game, where the player controls the launch of a small projectile from the bottom of the screen. The object that launches the projectile is slowly rotating from left to right, making the game primarily about choosing when to press the button in order to control the trajectory of the projectile. A launched projectile bounces off the side of the screen and slows down according to a model of physical drag. When it stops, it expands in size until it collides with either the side of the screen, or with another stopped projectile. Stopped projectiles display a number, and this number decreases whenever a projectile collides with it. When a stopped projectile's number hits zero, the stopped projectile is removed from the screen and the player's score increases. The goal of the game

is to earn a high score.

It is clear that *Orbital* is a game with considerably more complex dynamics than *Kaboom!* While *The Game-O-Mat*'s engine is able to represent *Orbital*, it takes 419 lines of expanded JSON (roughly twice the size of Kaboom!), and this JSON is considerably more nuanced than that of *Kaboom!*, and it makes used of highly configured advanced features of the engine. To illustrate this, this section will describe some aspects of the *Orbital* projectile and its behavior.

First, projectiles are given a tag so that other components can target them. They also need to have two variables: one to represent the counter that decreases on collision (as part of the game's main mechanics), and another as a flag to track whether the projectile is in its moving state or a stopped state.

For the simple part of the component assignment, the projectiles need to have a "Forward Movement" component to make them shoot out from the launching turret, and a "Bounce" component to make them bounce off of other game objects with the projectile tag. Things start to become more complicated because the projectiles behave differently whether they're moving after being launched, or expanding, or stopped. Representing these different states required making use of the conjunction activator, and making the conjunction conditional on a property activator that checks the value of one of the game object's variables. For example, we need to make sure we only have the projectile bounce off the left and right walls when it is moving. If we didn't do this, when the projectile was done expanding (and likely touching a wall), we would be constantly activating the component that reverses the x velocity which would cause strange movement when the projectile should be staying still. Instead, we make the component that modifies the velocity first target the left and right walls, and then make use of a conjunction activator with both property activator checks seeing if the projectile is moving or not, and also a normal collision activator. In order for this to work, we need to change the value of the moving variable to false at a different area of the game description.

Orbital also necessitated the use of a Scored Rule Engine (SRE) type activator in order to decrease a projectile's counter. In English, each projectile checks the following four conditions: Am I colliding with something (Colliding)? Is that something another projectile (HasTag)? Is that other thing moving (GameObjectPropertyCheck)? Am I not moving (GameObjectPropertyCheck)? If the answer is yes to all four questions, activate the "ModifyGameObjectProperty" component and decrement the counter variable of the game object.

These are just two of many nuanced and interconnected uses of the engine that were required to create *Orbital*. The purpose of these examples was to demonstrate the point at which the structure of this engine limits its expressivity. While the generator can in principle create a game with dynamics like *Orbital*, it is unlikely.

This example also shows how more committed and specific components could be authored to enable certain sets of mechanics (i.e. the behavior of an *Orbital* projectile could in principle we a single component). Situations like these are important to consider when creating a game engine aimed at generation. In the case of *The Game-O-Mat*, it is currently more of a priority to enable the generation of Atari 2600 style games, and no "Orbital Projectile" component will be introduced. However, the engine *does* support the representation of such games.

Conclusion and Future Work

This paper presented the game engine that the forthcoming *Game-O-Mat* is going to use to represent the games it generates. The goal of the paper is to document in detail some of the considerations and design choices made in order to create an engine that is configurable by an artificial intelligence system. Future work will of course involve the completion of the generator and the crafting of the user experience. In addition to being a media artifact that stands on its own, the goal of this project is to serve as a platform for research in component-based game generation.

References

Cook, M.; Colton, S.; and Gow, J. 2017. The angelina videogame design system—part i. *IEEE Transactions on Computational Intelligence and AI in Games* 9(2):192–203.

Duplantis, T.; Karth, I.; Kreminski, M.; Smith, A. M.; and Mateas, M. A genre-specific game description language for game boy rpgs.

Ivey, P.; Ferrer, B.; Saunders, R.; Gaudl, S. E.; Powley, E.; Nelson, M.; Colton, S.; and Cook, M. 2018. A parameter-space design methodology for casual creators. In *ICCC*.

Liapis, A.; Yannakakis, G. N.; Nelson, M. J.; Preuss, M.; and Bidarra, R. 2019. Orchestrating game generation. *IEEE Transactions on Games* 11(1):48–68.

Summerville, A.; Martens, C.; Samuel, B.; Osborn, J.; Wardrip-Fruin, N.; and Mateas, M. 2018. Gemini: Bidirectional generation and analysis of games via asp. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 14(1):123–129.

Treanor, M.; Blackford, B.; Mateas, M.; and Bogost, I. 2012. Game-o-matic: Generating videogames that represent ideas. In *Proceedings of the The Third Workshop on Procedural Content Generation in Games*, PCG'12, 1–8. New York, NY, USA: Association for Computing Machinery.

Wardrip-Fruin, N. 2020. *How Pac-Man Eats*. MIT Press.