

UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Mohamad Qadura

WebGLadiator Game Engine For Web Developers

Master's Thesis (30 ECTS)

Supervisor: Margus Luik

Tartu 2017

WebGLadiator Game Engine for Web Developers

Abstract:

With the modern WebGL technology, the adoption of web games has increased drastically leaving a gap between the low number of available WebGL developers and high demand for them. Since WebGL resides in JavaScript ecosystem, developers are more likely to come to WebGL with web development background. In the JavaScript ecosystem, developers are accustomed to JavaScript for coding, HTML for structure, and CSS for layout and design. The present game engines built on top of WebGL do have a set features to develop games, however, they do not take JavaScript Ecosystem into consideration which makes it hard for developers with web development background to migrate to web games development. WebGLadiator is a game engine for web developers that facilitates the migration from web development to game development by providing the same approach used for web development. In order to keep the same approach, JSON, used as blueprint files, will be used instead of HTML to structure games, VFL will be used instead of CSS to layout them, and JavaScript remains the programming language to write the logic. In this project, we are going to use open source libraries that will inter-operate under one game engine following proper software design patterns and architecture to create a hybrid ECS that utilizes the ECS architecture to provide the same approach for web development.

Keywords:

WebGL, Game Engine, Software Design Patterns

CERCS: P170

WebGLadiator Veebiarendajatele mõeldud mängu mootor

Lühikokkuvõte:

Kaasaegse WebGL tehnoloogiaga, veebimängude juurutamine on meeletult kasvanud, luues lõhe WebGL arendajate nõudluse ja pakkumuse vahel. Kuna WebGL töötab JavaScripti baasil, võib suure tõenäosusega näha WebGL arendusega liitumas veebitehnoloogiate taustaga arendajaid. JavaScripti ökosüsteemis on arendajad harjunud JavaScriptiga koodi osas, HTML-ga struktuuri osas ja CSS-iga välimuse ja disaini osas. Praegused mängumootorid on ehitatud WebGL baasil ja omavad mängudeks selleks ettenähtud funktsionaalsust, kuid need ei võta arvesse Javascripti Ökosüsteemi, mis raskendab veebiarendajate sisseelamist veebimängude arendusse.

WebGladiator on mängumootor veebiarendajatele, mis hõlbustab veebiarendajate liikumist mänguarendusse, pakkudes sama lähenemist, nagu veebiarenduses. Selleks, et hoida sama lähenemist, JSON on võetud kasutusse HTML-i asemel struktuuri tehnoloogiana. VFL tehnoloogiat kasutatakse CSS-i asemel paigutuste ja disaini jaoks ning JavaScript jääb programmeerimiskeeleks, milles kirjutatakse loogika. Selles projektis me kasutame avatud lähtekoodiga teeke, mis töötavad koos ühes mootoris, järgides õigeid

tarkvaraarenduse mustreid ja arhitektuuri, et luua hübriid ECS, mis utiliseerib ECS arhitektuuri, pakkumaks sama lähenemist veebiarenduse jaoks.

Võtmesõnad:

WebGL, Mängu mootor, Tarkvara disainilahendused

CERCS: P170

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 7 |
| 1.1 | Game Engine Architecture | 8 |
| 1.1.1 | Graphics | 9 |
| 1.1.2 | Sounds | 11 |
| 1.1.3 | Physics and Tweens | 12 |
| 1.1.4 | HID | 13 |
| 1.1.5 | Resource Management | 13 |
| 1.1.6 | Communication | 13 |
| 1.2 | Available Technologies | 13 |
| 1.2.1 | Programming Language | 13 |
| 1.2.2 | Ecosystem | 15 |
| 1.3 | Added Technologies | 15 |
| 1.3.1 | Chrome Extension | 15 |
| 1.3.2 | Dev-Hubs | 16 |
| 2 | Methods | 17 |
| 2.1 | System Design And Architecture | 17 |
| 2.1.1 | Entiy/Component System | 17 |
| 2.1.2 | Managing Life Cycle of Entities | 20 |
| 2.1.3 | Integral Graphics and Layout | 21 |
| 2.1.4 | Managing External Libraries | 22 |
| 2.1.5 | System Architecture | 23 |
| 2.1.6 | Messaging Buses | 24 |
| 2.1.7 | Solidarity Using Observer Pattern | 24 |
| 2.2 | Web Approach | 25 |
| 2.2.1 | Blueprints | 26 |
| 2.2.2 | Layout with VFL | 27 |
| 2.2.3 | JavaScript | 29 |
| 2.3 | Reusability | 29 |
| 2.3.1 | Reskinning | 29 |
| 2.3.2 | Reusable Entities | 30 |
| 2.3.3 | Registry | 30 |
| 2.4 | Tooling | 31 |
| 2.4.1 | Chrome DevTool | 31 |
| 3 | Results | 33 |
| 3.1 | Walk Through | 33 |
| 3.1.1 | Registering Systems | 34 |
| 3.1.2 | Game Blueprint | 35 |

| | | |
|----------|---|-----------|
| 3.1.3 | Scene Blueprint | 37 |
| 3.1.4 | Logic | 38 |
| 3.1.5 | Scene Management | 39 |
| 3.1.6 | Playing The Game | 40 |
| 3.2 | Web Tailored Engine | 42 |
| 3.2.1 | Web Technologies | 42 |
| 3.3 | Agile Development | 43 |
| 3.3.1 | Easy Game Setup | 43 |
| 3.4 | Organized Work flow | 44 |
| 3.4.1 | Coherent Engine | 44 |
| 3.4.2 | Readability | 45 |
| 4 | Discussion | 47 |
| 4.1 | Comparison | 47 |
| 4.1.1 | List of Engines | 47 |
| 4.1.2 | Comparison between available engines | 48 |
| 4.1.3 | WebGLadiator | 52 |
| 4.2 | Decision Making | 52 |
| 4.2.1 | Did I reinvent the Wheel? | 52 |
| 4.3 | Critics | 53 |
| 4.3.1 | Competing Other Engines | 53 |
| 4.3.2 | Software Design Patterns for Games | 53 |
| 4.3.3 | Adoption Risk | 53 |
| 5 | Conclusion | 54 |
| 6 | Future Work | 55 |
| 6.1 | Modularity | 55 |
| 6.1.1 | Dependency Injection And Inversion of Control | 55 |
| 6.1.2 | Game Builder | 55 |
| 6.2 | New Features | 56 |
| 6.2.1 | XAML | 56 |
| 6.2.2 | Unobtrusive VFL | 56 |
| 6.3 | Open Source | 56 |
| 6.3.1 | Non-Disclosure | 56 |
| 6.3.2 | Documentation | 57 |
| | References | 59 |

| | |
|-----------------------|-----------|
| Appendix | 60 |
| I. Glossary | 60 |
| II. Licence | 61 |

1 Introduction

The rapid growth in the mobile industry, as well as the expansion of low-end and small-sized devices, has made 2D gaming market a heavily demanded trend. Compared to 3D games, 2D games are relatively simple to develop and maintain as well requires fewer resources to run on devices. Hence, while the demand for 2D games was dropping down as a result of the growth in 3D games, the introduction of low-end devices was the major factor to bring back 2D games to the scope.

Casual game genre, which we are going to focus on for this project, is One of the most trending genres of 2D games because it is easy to learn and play, friendly for any age and gender, and less demanding in terms of time that motivated players of all ages and genders to play. Minesweeper, Tetris, and FarmVille are examples of famous casual games, as well as Candy Crush the famous game that was played by over 93 million people and reported a revenue of 493 million dollars in three months.

The trend in 2D gaming along with the spread of casual games influenced Facebook early 2017 to start their gaming platform 'Instant Games'. Instant games platform is more convenient for casual games stressing out the ease of access and availability for developed games. The new demand for low-end games is continuously motivating gaming companies to move towards 2D gaming since it is the lightest form of a game.

A game engine is a software framework designed for the creation and development of video games. Developers use them to create games for consoles, mobile devices, and personal computers. The core functionality typically provided by a game engine includes a rendering engine ("renderer") for 2D or 3D graphics, a physics engine or collision detection (and collision response), sound, scripting, animation, artificial intelligence, networking, streaming, memory management, threading, localization support, scene graph. The process of game development is often economized, in large part, by reusing/adapting the same game engine to create different games, or to make it easier to port games to multiple platforms[War].

Game development companies specialized in web gaming industry start by choosing an available open source game engine. After that, they start wrapping and customizing it with their own in-house code. In other words, they do not reinvent the wheel, they just reshape it accordingly. The reason behind this approach is to achieve ultimate control on the engine used and drive it towards their ultimate goal of making it as controllable and easy as possible.

While this approach indeed reduces the time spent in making games, as well as the LOCs required to be added for new games, companies still are not able to bring a game engine that would attract web developers to develop web games. Considerably, WebGL is still a new technology that runs relatively slow on low-end devices, leaving developers open to developing low-end games e.g 2D games for mobile phones. New developers who are willing to learn game development would then rather learn how to make high-end 3D games than to stay with 2D games, for example, they might start

with Unreal or Unity leaving a gap for WebGL technology. Our Project, WebGLadiator, will fill that gap for Web developers and simplify the process of developing games by standing as a bridge between game development and web development.

Nowadays, there are plenty of available open source game engines, as well as different libraries that can be put together to build a game engine. Depending on developers' choice they can either use an available engine or build their own. For this project, we will be using open source libraries, modifying them if need be, and finally, combine them all into one game engine.

WebGLadiator will facilitate the migration for web developers by introducing the concept of blueprints which are a simple JSON file that will be used instead of HTML, VFL will be used instead of CSS while keeping JavaScript the programming language. Moreover, it will as well present a capable engine with its features. Beyond being oriented to the web, WebGLadiator will add extensibility, responsive design, and readability.

1.1 Game Engine Architecture

The core layer of a game engine is a set of systems denoted for functionality where each system is responsible for an exclusive functionality e.g, graphics, sounds or resource management. Although each system is responsible for its own functionality, they have to communicate with each another and the game itself. Game Loop, the heart of a game, is the central communication between the systems e.g when to load resources and render them, or when to play sounds and stop them. Figure 1 illustrate its architecture.

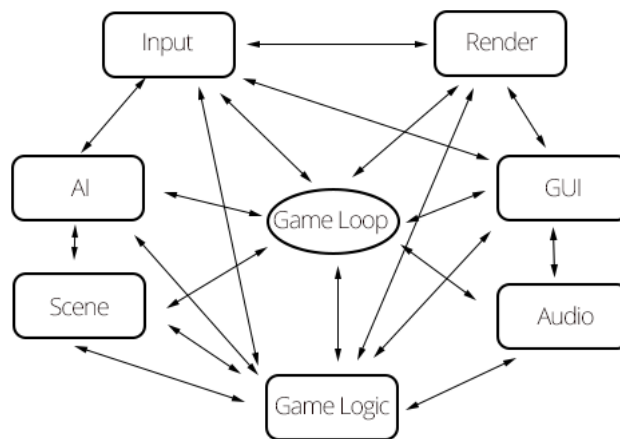


Figure 1. Game Engine Architecture

The game loop is the center code of the game. Characterized by a state that changes based on contemporary state and different variables, game logic. Game loop can have 3 main states with a non-mandatory state for resource loading :

1) Preload: the state in which required resources are being loaded, a game or otherwise a scene cannot be created without the required resources being loaded.

2) Initialize: This is the state where initialization takes place and variables are assigned their values. Moreover, information gathered from screen size and ratio, input devices, and operating systems will be used in this state.

3) Update: The main purpose of the update phase is to prepare all objects to be drawn, so this is where all the game changes such as coordinate updates, health points changes, damage dealt, as well as physics recalculations. This is also where the input will be captured and processed.[Mol12]

4) Draw: This state is where all current variables and states get translated into meaningful graphics.

1.1.1 Graphics

A picture is worth a thousand words. Graphics visualize to the player the current state based on all current variables. We can imagine the score, a display object moving on the screen or even a menu button that might or not be disabled. Graphics are commonly considered the core functionality of a game because it is, after all, what the player sees while playing a game.

For this project, we are going to use Pixi.js as the graphics system for 2D rendering. Pixi is a rendering library that will allow you to create rich, interactive graphics, cross platform applications, and games without having to dive into the WebGL API or deal with browser and device compatibility[Dig]. In addition to Pixi, AutoLayout.js will be used as a layout engine, AutoLayout.js is an abstract library for integrating Auto Layout and VFL into other javascript technologies. It provides a simple API and programming model that you can use to build your own auto layout and VFL solution[Rut]. VFL for WebGLadiator is what CSS is for web sites.

By design, Pixi's default behavior for containers is that they scale when resized causing all contained children to be scaled as well, with this behavior Autolayout will cause resized children to reposition and resize which leads them to be different from what we expect. This could only be permitted for games with aspect-ratio enforced. Figure 2 is the code snippet from Pixi responsible for scaling the container when changing its dimensions

```
575     set width(value) // eslint-disable-line require-jsdoc
576     {
577         const width = this.getLocalBounds().width;
578
579         if (width !== 0)
580         {
581             this.scale.x = value / width;
582         }
583         else
584         {
585             this.scale.x = 1;
586         }
587
588         this._width = value;
589     }
```

Figure 2. Pixi Snippet for Scaling

Since we want to have responsive games that adapt to any screen size, aspect-ratio will not be a restriction for us. Even though a full window game does not have an aspect ratio, sub components inside the scene do have aspect ratios. For that reason we introduced two types of containers:

1) Spacing Container: this container will never scale nor resize, it will instead pass the boundaries to its children so that they know where to position themselves.

2) Scaling Container: this container will act like a Spacing Container in the Create state, however, upon a render state it will only scale and stop its children from being updated.

The combination of both types of containers will lead to a seamlessly responsive game that fits fully into any screen regardless of the aspect ratio. Figure 3 illustrates how an inner aspect ratio of the smiley face can be reserved regardless of the view port aspect ratio

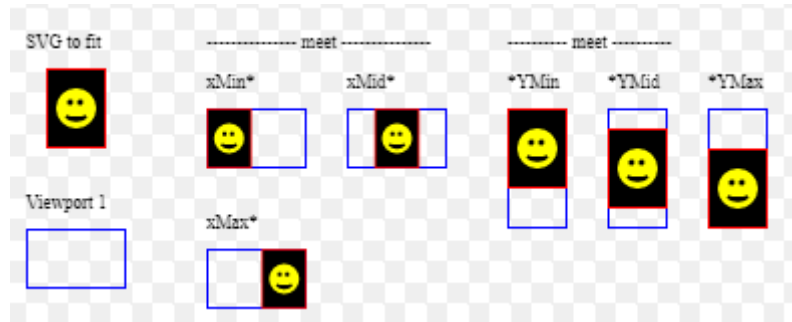


Figure 3. Reserving Inner Aspect Ratio

1.1.2 Sounds

The sound system usually responds to events by the user and plays a sound as an indication for a user's action e.g user click sound. Moreover, sounds can play as a background music hence a game can start while the music file is being lazily loaded.

Since Pixi has a resource loader for assets yet not sound, we will use Pixi-Audio middleware to add a loading parser for sound files. In the future, all Pixi loaders will be replaced by custom loaders and middlewares on top of another resource loading library "resource-loader". The purpose is to achieve more flexibility in the code by reducing dependency on Pixi and move toward an easily customizable sound system.

1.1.3 Physics and Tweens

Physics is better explained with a real life example like Angry Birds game. When the player pulls the sling to a specific angle and a specific distance then releases his finger, the physics system calculates based on the distance, angle, bird's weight, and gravity the resulting trajectory in which the bird should fly with acceleration 4 describes physics in Angry Birds Game

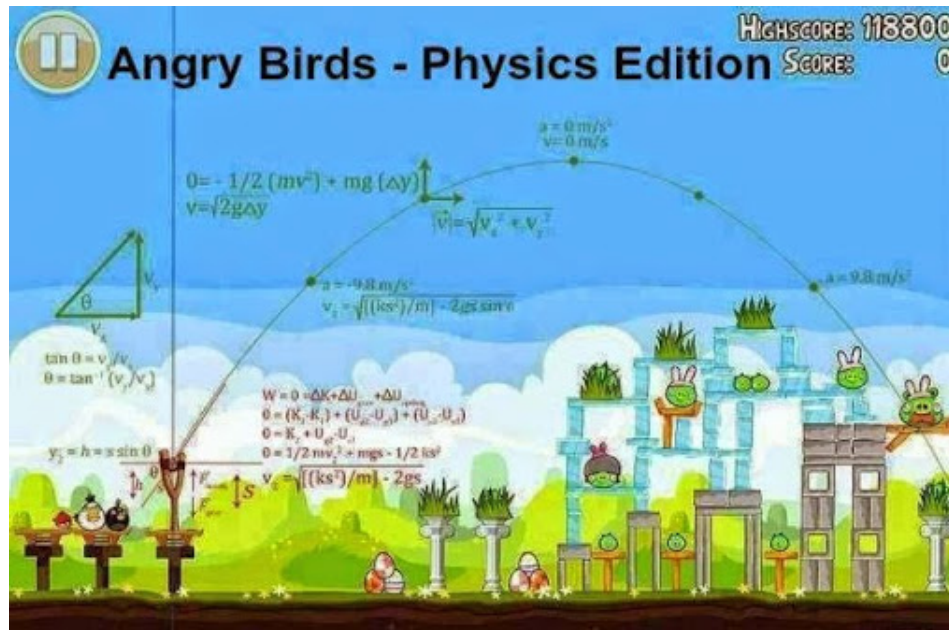


Figure 4. Angry Birds Physics

Tweens on the other hand only take time into consideration and add smoothing to it. When dealing with tweens, a developer can change the value of anything based on the current time covered e.g moving 200 pixels horizontally over 1 second. For our project we will be concentrating on Tweens rather than Physics since casual games, with Angry Birds exception, rarely require any physics. Tween.js will be used as a tween library.

1.1.4 HID

Human Interaction Device is what connects the player to a game usually, nowadays touch screens on mobile phones and tablets are vastly used. The main difference between a mouse and touch screen is that with the mouse the pointer is always available which means that it is always known where the user is currently pointing at. For touch screens, however, only when the user touches the screen the coordinate can be known. considering hovering as an example, with a mouse a user can simply hover by moving the mouse but with touch, the user has to implicitly tap.

Since Pixi handles graphics, any HID event that is bound to graphics is handled by Pixi as well e.g tapping a button. However, keyboard and gamepads are excluded from Pixi. In this project, we are going to add a unified layer for both mouse and touch devices and defer Pixi's EventEmitter with observer pattern using RX.js in order to simplify the process of differentiating between user actions. Keyboard support will be added in the future, as the initial game does not require a keyboard, only an abstract keyboard class will be used for now.

1.1.5 Resource Management

Pixi does support drawing custom graphics, but for an eye candy game using images is much preferable over drawing graphics. Not only will resource loading deal with images, it will also deal with sounds and JSON files.

1.1.6 Communication

Communication is the means of interaction between all the components in the engine as well as the game rules. Communication is event-based characterized by a command or a message sent when a set of event occurs for a component that meets a current condition.

1.2 Available Technologies

Nowadays, there are plenty of available open source technologies that will help build a project from scratch without the need to buy any license. However, with greater power comes greater responsibility, choosing the software is not an easy task, it has to be well planned and researched. Although the presence of many choices sounds to simplify the process, it can, however, make it even more complicated if wrong software were chosen.

1.2.1 Programming Language

Since WebGLadiator will target web games, for this project we are going to use TypeScript. Typescript is a super set of JavaScript that brings object-oriented programming to

JavaScript as well as typing. The mere benefit of Typescript is that it allows Object Oriented Programming which is much cleaner and simpler than the prototyping JavaScript. Moreover, since it is strongly typed, it is more understandable for both developers and can utilize the use of IDE. figure 5 Illustrates the difference between the languages

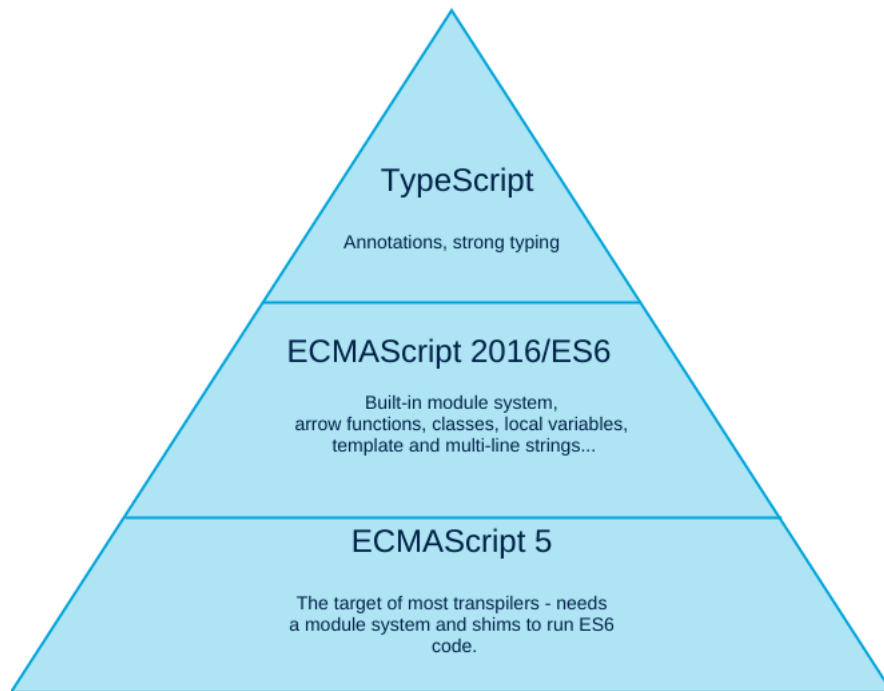


Figure 5. Javascript to Typescript

1.2.2 Ecosystem

In order to work with typescript and manage our libraries, we are going to rely on environment software that helps in development process while keeping in mind the future collaboration between different developers.

1) Visual Studio Code (VSCode): VSCode is a true example of extensible software, it allows addition of extensions, as well as configuring the environment as per our requirements, we can consider here auto-formatting which is a vital case in software development, imagine multiple developers working on the same file and everyone has his own way of writing code. Any developer who pushes/pulls his code will end up having multiple committed changes that are only related to formatting.

2) Git: a version control system for tracking changes in computer files and coordinating work on those files among multiple people. It is primarily used for software development, but it can be used to keep track of changes in any files.[AS]

3) Git Kraken: an open source GUI to utilize the use of Git that handles or the main operation to be done by Git and simplifies visually the process of handling repositories.

4) Microsoft Team Service: Formerly called Visual Studio online, is a service by Microsoft that utilizes the communication and collaboration inside a team of developers. Although this project is one be one person, the service is still vital as it offers a backlog to track the progress and the ability to create user stories, features, bugs, and utilizes Git-Flow.

5) Node.js: an open-source, cross-platform JavaScript runtime environment for developing a diverse variety of server tools and applications. Although Node.js is not a JavaScript framework, many of its basic modules are written in JavaScript, and developers can write new modules in JavaScript. The runtime environment interprets JavaScript using Google's V8 JavaScript engine.[Cuo]

1.3 Added Technologies

Since our project is not only about using available libraries and combining them all together, we are going to introduce a set of features that will contribute to the project and make development much easier and stable

1.3.1 Chrome Extension

Google allows the addition of extensions for chrome that can be used to interact with the browser hence an inspected window. Accordingly, we are going to do develop an extension and use to interact with the engine and all its systems, the extension will allow real-time editing and debugging of the running game built with the engine. Since we are using solid design patterns and we are managing life cycles behind the scenes no extra

code will be needed to make the extension work. Instead, the extension will access and read required data from the engine and display it for the developer.

1.3.2 Dev-Hubs

Dev-Hubs, created by Mohamad Qaddura, is a developer-to-developer approach in E-Learning built around the flexibility of GitHub and Open Source universe. Hubs address the hardship in the learning curve in complicated yet useful technologies and the complexity of acquiring or otherwise finding the proper resource to study.

Current available Hubs that relate to our project are ReactiveXHub and ChromeExtensionHub. If need be, any complicated technology used in this project will as well has its own Hub. Never the less, VFL is a very hard to learn and there is no enough resource on how to use with Autolayout. Hence, at some point, VFLHub should be added.

2 Methods

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations [Shv]. In this project, we are going to rely heavily on software design patterns not only for solving problems from our perspective but also from developers' perspective making it easier to understand the conventions we use.

Moreover, we will as well use the same convention as with other game engines to be coherent. However, we will improvise in our architecture and take slightly different methods to make the engine simpler and more specific, in other words, we are going to make the engine more specific towards casual games while maintaining a level of flexibility for the engine to be used, in the future, for other types of games.

In our project we will be building a Tic Tac Toe game as a showcase for the engine, hence most of the examples given will be based on the game itself.

2.1 System Design And Architecture

WebGLadiator uses software design pattern along with abstract level from an Entity/-Component System derived from the component design pattern. The use of software design patterns with the focus on the component pattern will help simplify the process for a developer. Hence, each design has its own responsibility, while other software design patterns are generic and can be applied to any software architecture, the component pattern is specific to game engines.

2.1.1 Entity/Component System

Component/Entity Systems are an architectural pattern used mostly in game development. An ECS follows the Composition over Inheritance principle to allow for greater flexibility when defining entities (anything that's part of a game's scene: enemies, doors, bullets) by building out of individual parts that can be mixed-and-matched. This eliminates the ambiguity problems of long inheritance chains and promotes clean design. However, ECS systems do incur a small cost to performance[Wik].

Einstein once said, "Everything should be made as simple as possible, but not simpler". In an ECS, creating different entities (Tic Tac Square) with different components (Sounds, Textures) means that our blueprint file will grow larger in size since we have to also include the components to be used. Hence, it is reasonable to have a lightweight ECS in order to shorten the time of learning as well as building a simple game. An ECS alone would add a complication for developing casual games since the developer will have to get familiar with the architecture before starting. Moreover, developers coming from

Pixi or Phaser background are not used to ECS and that would require extra tutoring and teach for developers. In terms of Pixi, figure 6 illustrates the hierarchy of Display Objects where container acts as both an Entity and Display object.

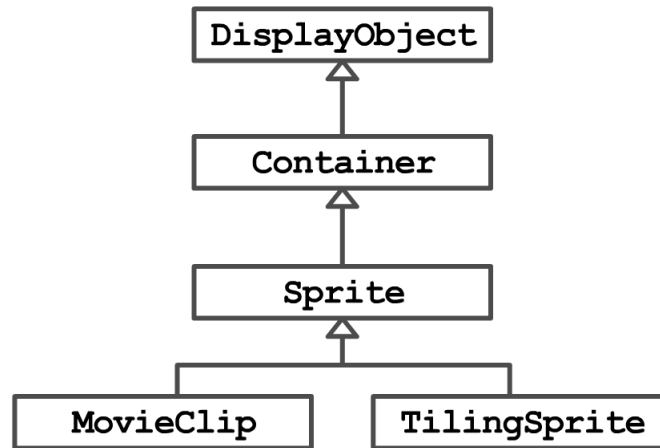


Figure 6. Pixi Hierarchy

Our system will slightly differ from the typical ECS in the way we compose entities components. Entities composition will be similar to that of standard ECS approach and facilitated by blueprint files. On the other hand, components will be implemented differently such that the entity will have direct access to the components e.g graphics and layout system will be an integral part of the entity. On the other hand, other components like sounds will access by entities based on the code logic since most sounds are played based on the occurrence of an event. Hence, In terms of an ECS, we are elevating entities to containers directly. 7 illustrates the hierarchy in WebGLadiator.

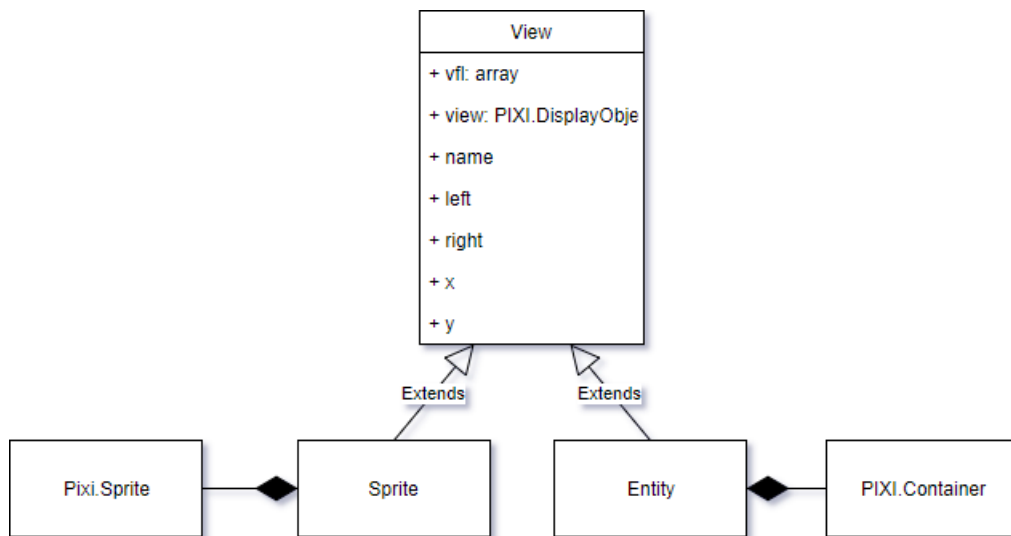


Figure 7. WebGLadiator Hierarchy

In the figure above, we use composition to add Pixi's display objects into our game objects as a twin. Our view class is the parent of all our game objects and entities and has all display properties, hence we right our display properties in the View class only without the need to do that in our inherited classes.

2.1.2 Managing Life Cycle of Entities

In terms of object creation, an object can either be directly created or lazily created when needed. Blueprints will utilize this process by using flags e.g isLazy flag which means that the lazy game object will not be created directly by the builder, instead, it will be created by code on demand. Moreover, a developer can use loops inside blueprint in order to create multiple instances of an object while applying different attributes for each instance.

The life cycle of a game object starts when it is it gets created by the blueprint builder. Blueprint builder uses builder pattern in order to minimize the number of arguments passed to the constructor and put them inside a blueprint file, this automates the creation of objects existing in the blueprint itself, afterward, the builder will initialize and execute each class's code without any interaction from the developer. Afterward, the initialization phase starts, and at this phase, variables are assigned their equivalent values e.g uncommon configuration in a blueprint. Once the objects are created and initialized, they start subscribing to messages, HID events. Figure 8 Illustrates the life cycle of objects.

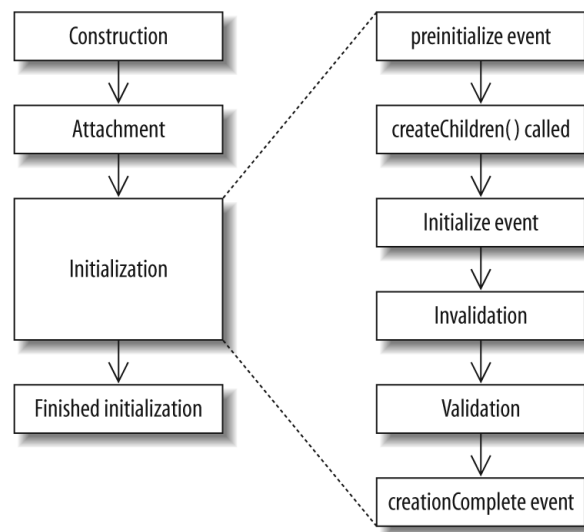


Figure 8. Life Cycle

The life cycle ends with the destruction of objects that are not needed anymore e.g when switching between scenes. Destruction sequence has opposite direction of the creation sequence such that objects first get unsubscribed, then variables are nullified, followed by the removal of the objects from the scene graph to avoid consuming or other wise leaking memory.

2.1.3 Integral Graphics and Layout

The layout engine is a constraint based engine that returns different calculated values for different screen sizes, objects will take their layout positions from result calculated by the layout engine. Moreover, objects will have dynamic positions changing by animations, yet these positions do not affect the result from the layout engine. For that reason, objects will have two displacement properties each, left is for the static displacement calculated from layout engine and x is for dynamic displacement calculated from animations e.g object sliding in. As a result, this will retain the layout of the game while allowing objects to animate.

Twin pattern is a software design pattern that allows developers to model multiple inheritances in programming languages that do not support multiple inheritances. This pattern avoids many of the problems with multiple inheritances[Mos]. Since JavaScript does not allow multiple inheritance, it is reasonable to use twin pattern, in our case in which will be applied to PIXI classes. This will help us compose PIXI classes and integrate it into our class as shown in figure 9 for our animated class.

```
9      constructor(owner, params) {
10          super(owner, params);
11          var frames = [];
12          params.frameList.forEach(frame => {
13              for (let i = frame.start; i <= frame.end; i++) {
14                  var val = i < 10 ? '0' + i : i; //TODO:use MathUtil
15                  frames.push(PIXI.Texture.fromFrame(`${frame.prefix}${val}${frame.postfix}`));
16              }
17          });
18      }
19      this.view = new PIXI.extras.AnimatedSprite(frames);
20  }
21  play() {
22      this.view.play();
23  }
24  pause() {
25      this.view.pause();
26  }
27  stop() {
28      this.view.stop();
29  }
30  gotoAndStop(frameNumber=0){
31      this.view.gotoAndStop(frameNumber)
32  }
33  }
34  }
35  }
```

Figure 9. Twin Pattern For Animated Sprite

The above figure helps us to apply multiple inheritances for our animated sprite and to be less dependant on code changes from PIXI. Hence, if in the future PIXI decided to rename the function `gotoAndStop` to `playUntil` we do not do a global change in the code, instead, we will use the new function from PIXI without the need to rename ours.

Now that we have our own twin classes we have to make them accessible by the layout engine. The layout engine is a simple iterator that traverses the scene graph from top to bottom and computes the dimensions and positions for all entities.

2.1.4 Managing External Libraries

When we add external libraries we have to keep in mind that they might get updated anytime. Another case is that we might want to use different libraries with same functionality e.g changing from one sound library to another. In order to integrate the system, we use adapter design pattern that used to allow an existent library to be used without modifying its source code by adding an interface to use it. Using this approach we ensure that we create a single system for similar purpose libraries, the pattern is illustrated in figure 10.

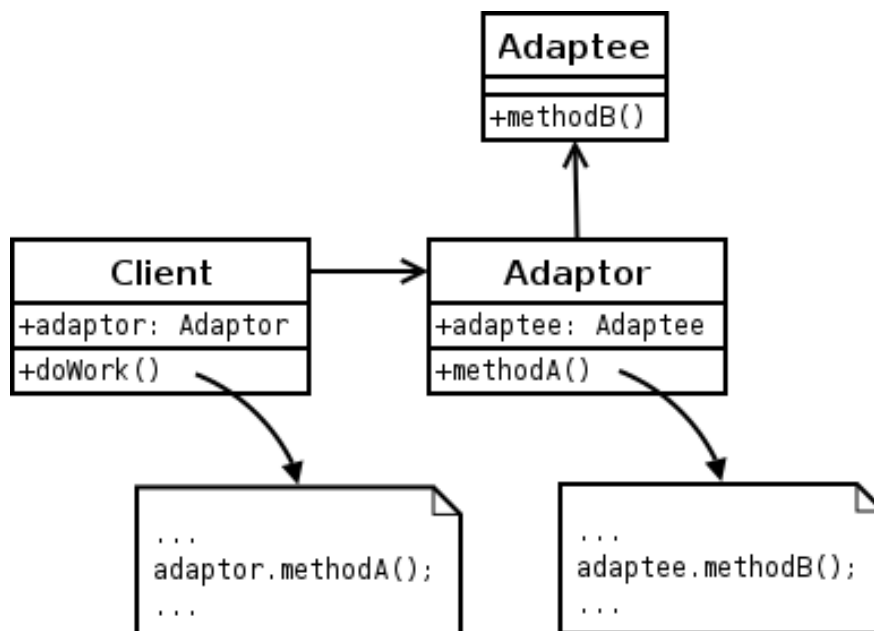


Figure 10. Adapter Design Pattern

We will take PixiAudioAdapter as an example here, whenever we want to play, pause, stop, or even change the volume of a sound we call the methods using our adapter which provides adequate access to our sound library while hiding the libraries implementation. In the future, if a used library was updated, then we only change our adapter to adapt the new changes without having to change the code anywhere else.

2.1.5 System Architecture

In order to separate the concerns of each functionality in the engine we are going to divide them into different systems e.g graphics system, sound system, and any other distinct functionality included. Over each system, we will use facade design pattern that utilizes access to a system and keeps unified code among similar functionality adapters.

Going back to our sound system example, even if we have two different sound adapters we will still use one for a game, the choice to which adapter to use is up to the developer. However, the facade will govern the access to the adapters to hide their implementation and isolate any functionality that is not a part of adapter's functionality, e.g autolayout adapter does not have to know about the current screen size, instead, the layout facade's takes responsibility. Accordingly, the developer will use the facade to access the corresponding adapter. Hence, this way we only need to change the adapter if we needed a different library. Facade pattern is illustrated in figure 11.

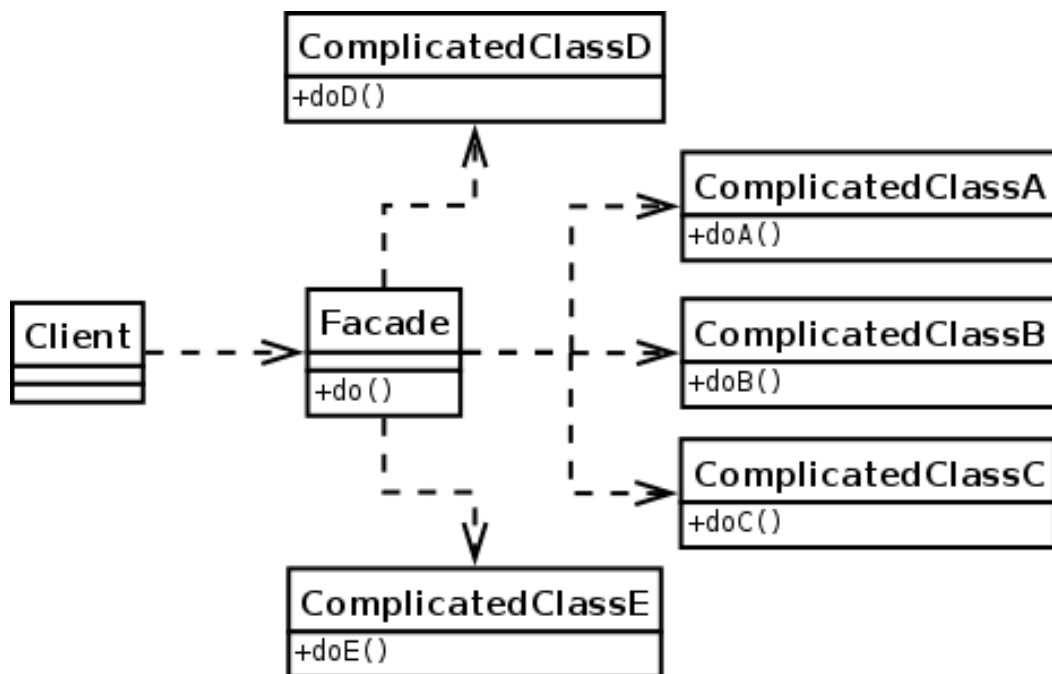


Figure 11. Facade Design Pattern

2.1.6 Messaging Buses

In order for objects to exchange events and information, we are going to use messaging buses to transmit messages between them. Messaging bus responsibility is not limited to transmit messages, it also will ensure that only components registered to a message will receive in order to avoid heavy communication and filtering.

In Javascript, we can implement communication using event emitter, signals, or observable stream. Table 1 gives a quick overview of their differences.

| Implementation | Favors | Event Type | Extensibility |
|----------------|--------------|------------|---------------|
| Event Emitter | Inheritance. | String | Limited |
| Signals | Composition | Member | Limited |
| Observer | Both | Object | Flexible |

Table 1. Communication using Event Emitter vs Signal vs Observer

As per our project, we want to obtain messaging that is composite and flexible, for that reason we are going to use observer pattern with RxJS. RxJS provides robust operations using operators that will simplify the code when dealing with multiple messages.

2.1.7 Solidarity Using Observer Pattern

Since games run over a set of events, messages, and operations we are going to use the observer pattern with ReactiveX. ReactiveX is a library for composing asynchronous and event-based programs by using observable sequences. It extends the observer pattern to support sequences of data and/or events and adds operators that allow you to compose sequences together declaratively while abstracting away concerns about things like low-level threading, synchronization, thread-safety, concurrent data structures, and non-blocking I/O [Mic].

to interconnect them together in a form of operation stream, this gives the flexibility to handle different kinds of events as shown in the figure12


```

1 Rx.Observable.fromEvent('click', canvas) //subscribe HID event
2 .bufferWithTime(1000) //Buffer the event over 1 second
3 .take(5) // take only 5 buffers
4 .reduce((accumulator, arr)=>{ //accumulator
5     return accumulator + arr.length; //add to accumulator
6 }, 0) //startWith 0
7 .subscribe((currTotal)=>{ //Recive the current total
8     doHeavyOperation(currTotal); //do heavy work
9 }),
10 (err=>{console.error(err)}),
11 (()=>{ console.info('completed')});
12 );
13

```

Figure 12. ReactiveX for Speed Click Game

In figure 12 we use ReactiveX to listen to a click event on the canvas, then we buffer all the clicks over one second interval. After that, we accumulate the total and then send the total to the heavy Operation. This allows us to interconnect HID events with time events inside a single stream. Hence, we did a complicated logic that would require otherwise to use different functions for each event, HID and time, and then another function to combine the calculations.

2.2 Web Approach

In order to provide a smooth migration from web development to game development, we will be focusing upon making the process of developing a game on our engine similar to what web developers are used to developing websites. In terms of web development, GUI is rarely used by developers, as developers see every single line of code and interact to it via text editors rather than IDEs and CLI instead of GUI software. But before we explain our criteria in making the engine oriented for web developers, we have to answer the firstly asked question, why do we have to use WebGL?

Until recently, most companies were using HTML and Flash to develop games. With the introduction of CSS3 and the enhancement HTML5 as well as the declination of Flash, companies started moving towards HTML5 which has proved itself capable of making simple games. Adding to that, HTML5 is cross platform and all it requires is a browser, which eliminates the need to install a game and that influenced the web games market. Moreover, CSS3 was a big advantage to games in HTML as it is robust and flexible in animations which make it easy to design and layout a game. On the other hand, HTML was meant for website not for graphics, which means not only performance will be affected but there are missing features that cannot be done with HTML, to mention simple yet effective features we can consider Geometry, which in HTML is absent hence all game objects come in rectangular shapes which become a clear inconvenience for HID. Adding to that, layering, blending, and masking is absent in HTML. Of course,

there is always a work around to achieve these features but that all comes at the cost of performance.

In order to void the costly work around required to develop games with HTML, we are going to use WebGL that is specific for graphics and includes geometry, blending, layering and masking. We will make the development on top of our engine similar to that of web development. For that purpose, we are going to introduce the concept of blueprints which are simple JSON files that will act just like HTML does in building the content the game and structuring it. VFL will be the replacement for CSS and JavaScript remains the programming language.

2.2.1 Blueprints

Blueprint concept is introduced to act like HTML so that we structure the content of the game in a readable way. instead of using HTML we are going to use JSON for the reason that it is easier to parse and manipulate with JavaScript. Moreover, we will use utilize operators inside in the blueprint inspired by Angular. for example, repeating, conditionals, filler. A blueprint file will describe the structure of the scene graph where all nodes can have children nodes that are entities, game objects, or components. Blueprint files can as well have nested blueprints inside them, figure 13 is a blueprint for a Tic Tac Toe Game

```

1 import { GameStateStrategy } from './GameStateStrategy';
2 import { GameplayStrategy } from './GameplayStrategy';
3 import { BlueprintBuilder } from './WebGLadiator/src/Foundation/Builder/BlueprintBuilder';
4 import { MainSceneBlueprint } from './MainScene/MainSceneBlueprint';
5 import { IntroSceneBlueprint } from './IntroScene/IntroSceneBlueprint';
6 import { OutroSceneBlueprint } from './OutroScene/OutroSceneBlueprint';
7 import { Game } from './Game';
8
9 export const GameBlueprint = {
10   name: "Game",
11   ctor: Game,
12   vfl: [
13     "HV:|[MainScene(100%)|",
14     "HV:|[IntroScene(100%)|",
15     "HV:|[OutroScene(100%)|",
16   ],
17   blueprints: [
18     MainSceneBlueprint,
19     IntroSceneBlueprint,
20     OutroSceneBlueprint,
21     {
22       "name": "GameStateStrategy",
23       "ctor": GameStateStrategy
24     },
25     {
26       "name": "GameplayStrategy",
27       "ctor": GameplayStrategy
28     }
29   ],
30   sceneMap: {
31     "MainScene": MainSceneBlueprint,
32     "IntroScene": IntroSceneBlueprint,
33     "OutroScene": OutroSceneBlueprint
34   }
35 }
36

```

Figure 13. Tic Tac Toe Game Blueprint

The game blueprint includes blueprints for all three scenes blueprints that has entities and components inside to use, and two strategies responsible for game overall logic throughout all the scenes. We can as well see VFL for every scene. sceneMap in this blueprint is a property of Game class, which can be passed by the blueprint to the class upon construction. This means that attributes, properties as well as configurations can be passed via blueprints.

2.2.2 Layout with VFL

CSS is robust and clean, using it for games to some extent can be helpful and easier for developers to use since they are already used to it. However, CSS can only be accessed by DOM elements which means in order to read CSS properties we have to have a complete DOM of the scene graph and that alone means we have to as well build HTML tree for

the game, for this reason using CSS will be an overhead to use. Hence, we will be using Visual Formatting Language (VFL) the language is used to minimize the constraints into small strings that would otherwise be large and hard to change. Layout system uses VFL as a language and autolayout.js as a library to parse the constraints passed as VFL. in the left of figure 14 we can see the used VFL, to the right is the result obtained.

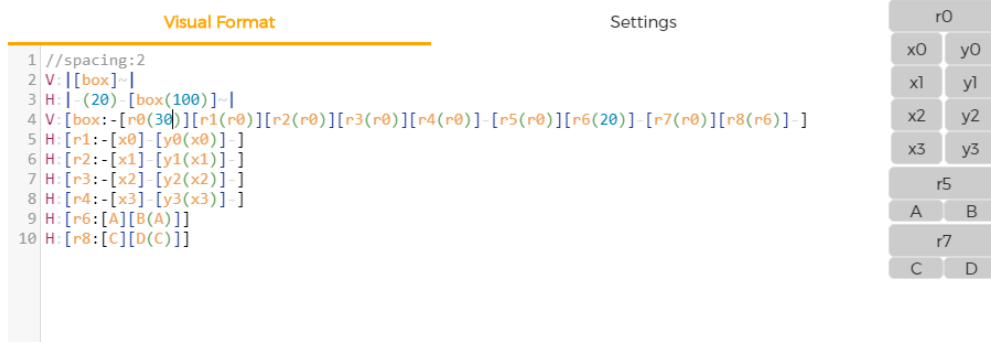


Figure 14. VFL Showcase

In order to make Games responsive, we will force aspect ratio by using VFL constraint, this way entities will resize based on the aspect ratio they are given inside the game as in Figure 15 shows both VFL and aspect ratio enforcement

```

41         "vf1": [
42             //"/viewport aspect-ratio:1/1",
43             'HV:|~[Grid(<=98%)]~|',
44             //'C:Grid.height(Grid.width)',
45             'C:Grid.width(Grid.height)'
46         ],
47     ],

```

Figure 15. Forcing Aspect Ratio With VFL

In the figure above, we force to have the width equal to the height of the grid which means that grid is 1x1. Moreover, using autolayout we obtain extra values that we can use for padding, anchoring, and pivoting as we can get the width, height, x, y, right, bottom, and center of any calculated object.

2.2.3 JavaScript

In order to read the game logic, we need a programming language and since we are using web technologies JavaScript is the one option. However, we will use a superset of JavaScript which TypeScript that will be transpiled into JavaScript upon building the game. The reason for using TypeScript is that it has types which directly enables auto complete and makes the code easier to understand, moreover TypeScript favors Object Oriented Programming over Prototyping that plain JavaScript uses.

Blueprints are made with JSON because it is faster and easier to read and update with JavaScript, this will give us more flexibility in both writing and debugging the code. That means developers can as well access blueprints from browser's console to see if there is something wrong in the blueprint itself.

2.3 Reusability

When developing multiple games we have to keep in mind that there are plenty of usable code that should easily be extended. Moreover, there are many cases where we want the same logic yet only we want to change the UI. Using the blueprint files we simplify the procedures of creating and using different UI. Moreover, it will also be easier to change the logic, since it is simple to just add the new game object to the blueprint.

2.3.1 Reskinning

Reskinning is a process where app developers do not have to create a game from scratch. You take a pre-made source code of a game and change its design completely, to give it a brand new look. There are many benefits of reskinning pre-made codes such as saving development time, increasing revenue, minimizing risks and so on. This is a pretty good way for even a beginner app developer to test out the waters in the app business [Gam].

the most basic form of reskinning is just to change the assets that we are preloading e.g the PNG files. However, sometimes we want to add new resources without removing a new one e.g two buttons with different assets. In our case, it will be as simple as adding a reference in the manifest file and using it in the blueprint file.

2.3.2 Reusable Entities

Although the appearance of a game object might differ from one to another, the logic does not always have to. imagine having a button that has a different sprite for each state, and another button that has a different animation for each state. This means that the only thing that is actually changing is the state machine of the button, the sprites and animations are no more than the visualization of the state. The state machine of a button is shown in the figure16.

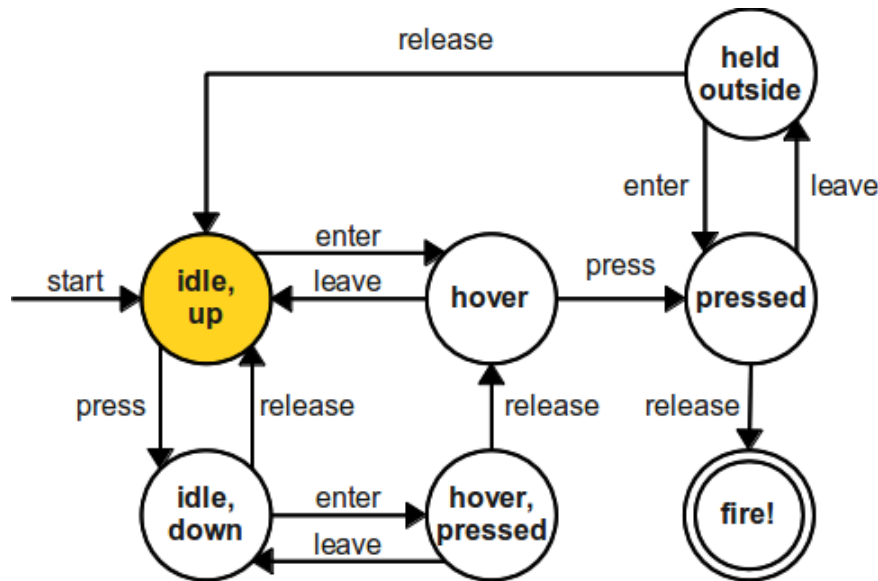


Figure 16. Button State Machine

For this purpose, we are going to have provided few different classes to be reused. That will include game objects e.g text, sprite, animated sprite. Moreover, Entities, are added including scene and application classes where the application is the root of the game and can contain different scenes. Adding to that our UI will include primitive like buttons and widgets that would be a menu and drop downs. Due to time restriction, only the folder for widgets is created without the implementation of the equivalent classes.

2.3.3 Registry

Since we want to separate the entities and controls that we provided from our engine code we are going to use a git submodule for that and include in it only the reusable entities without including anything from the system, this way it is much easier to get ready made game objects that are actually needed for a game.

2.4 Tooling

A programming tool or software development tool is a computer program that software developers use to create, debug, maintain, or otherwise support other programs and applications. The term usually refers to relatively simple programs, that can be combined together to accomplish a task, much as one might use multiple hand tools to fix a physical object. The ability to use a variety of tools productively is one hallmark of a skilled software engineer [Ker]. Our main tool for this project is the chrome devtool.

2.4.1 Chrome DevTool

A DevTools extension adds functionality to the Chrome DevTools. It can add new UI panels and sidebars, interact with the inspected page, get information about network requests, and more. View featured DevTools extensions. DevTools extensions have access to an additional set of DevTools-specific extension APIs[Goo] the workflow for a devtool extension that we are going to use is illustrated in figure 17

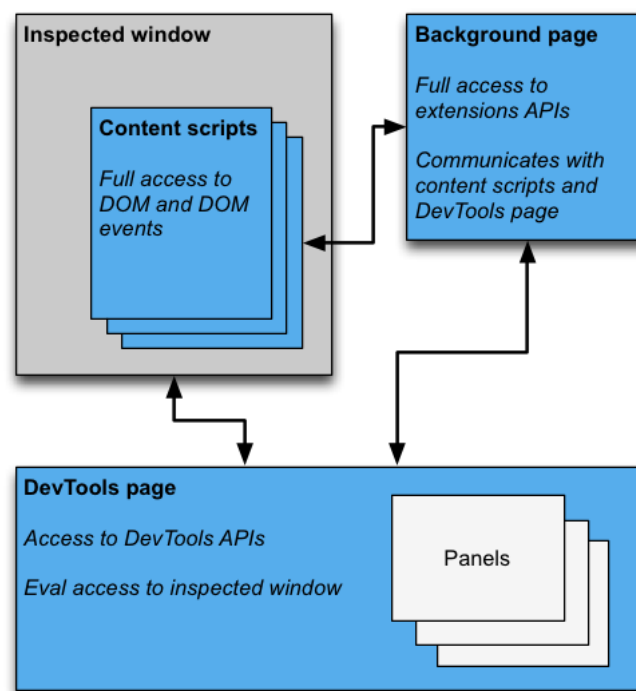


Figure 17. Chrome DevTools

JavaScript is a weakly typed prototyping language which means we have control over any object by just having a reference to it. Hence, we change the prototype of the object and treat it as a JSON object to call its methods, which we are going to use for our chrome extension, Figure 18 illustrates the simplicity of doing so

```
1 var something = {  
2   greet:function(name){alert(name)}  
3 }  
4 something['greet']('hello')  
5
```

Figure 18. JavaScript Method Calls

3 Results

The implementation of our engine provides many benefits for game developers that will use it. These benefits can reach out to a single developer as well as a team of developers. Initially, new web developers do not have to accommodate themselves to the game engine, as it will be used the same way they used to develop websites. Beyond that, If a developer wants to develop a game than our engine would make it as smooth as possible from start to end. Adding to that, if a company wants to make games and reuse what they did for previous games then they can make good use of our engine.

Not only will we make the process of developing games faster, we will also separate the concern between the UI and the logic as blueprints will handle most of the UI without risking control over the code logic as it goes the same way it is with web development where separation of concerns is well considered and maintained in all frameworks. Moreover, In the cases where developers want to change from one library to another, they can use their own adapter for that purpose. Altogether, developers will still have the same confidence as they have when developing websites, regardless of being concerned about their experience in game development.

Before we start with explaining our result, we are going to have a quick walk through on how to build a game using WebGLadiator. The walk through will be on how to build a Tic Tac Toe game with WebGLadiator.

3.1 Walk Through

In this section, we are going to build our Tic Tac Toe game on top of WebGLadiator. Accordingly, we will see how to register needed system and which adapters to use for each system, then how to divide our game into different scenes using blueprints. After that, we are going see how to nest blueprints inside scenes. Finally, we will check how to write the code to run our game logic. In order to keep it readable, we will write the game using a simple coding style that means we might have extra lines of codes that could be optimized especially when it comes to ReactiveX operators, that will be used as least as possible.

3.1.1 Registering Systems

Before we start building the game, we have to choose which systems we are going to use e.g whether to use a sound system or not. Accordingly, we have as well to choose the adapters equivalent to each library we want to use and inject it into the system. Our main script, first executed script in TypeScript, will be where we take this step and write our system related code there as shown in figure 19

```
23 let manifest = {
24   device:new DeviceAdapter(),
25   event:new Bus(),
26   layout:new AutolayoutAdapter(),
27   net:new ColyseusAdapter(`${"localhost"}:${3555}`, 'tictactoe'), //new WebSocketIOAdapter('localhost:3555'),
28   resource:new PIXIResourceAdapter(),
29   tween:new TweenJSAdapter(),
30   sound:new PixiAudioAdapter()
31 }
32
33 System.getInstance().inject(manifest);
34 BlueprintBuilder.getInstance().createObject(null, GameBlueprint);
35
```

Figure 19. Registering Systems

In the figure above we inject all our adapters into their equivalent systems. Later on, we can refer to these systems that will take responsibility of using the adapter without the need to have direct access to the adapters. It is worth mentioning, that during the development we decided to change from WebSocketIO, used for real time multiplier gaming, to Colyseus, which adds lobby management to WebSocketIO, and it was done by simply changing the injected adapter without the need to do global changes in the code.

At the bottom of the figure, we call blueprint builder to build our game blueprint which we are going to go through later. Hence, we build a game from the main script itself.

3.1.2 Game Blueprint

In order to build a game, we have to ensure the simplicity of both the process and structure. Game blueprint will be used to contain different scenes and every scene can have its own folder with required files. Figure 20 shows the game blueprint.

```
1 import { GameStateStrategy } from './GameStateStrategy';
2 import { GameplayStrategy } from './GamePlayStrategy';
3 import { BlueprintBuilder } from '../WebGLadiator/src/Foundation/Builder/BlueprintBuilder';
4 import { MainSceneBlueprint } from './MainScene/MainSceneBlueprint';
5 import { IntroSceneBlueprint } from './IntroScene/IntroSceneBlueprint';
6 import { OutroSceneBlueprint } from './OutroScene/OutroSceneBlueprint';
7 import { Game } from './Game';
8
9 export const GameBlueprint = {
10   name: "Game",
11   ctor: Game,
12   vfl: [
13     "HV:|[MainScene(100%)|",
14     "HV:|[IntroScene(100%)|",
15     "HV:|[OutroScene(100%)|",
16   ],
17   blueprints: [
18     MainSceneBlueprint,
19     IntroSceneBlueprint,
20     OutroSceneBlueprint,
21     {
22       "name": "GameStateStrategy",
23       "ctor": GameStateStrategy
24     },
25     {
26       "name": "GamePlayStrategy",
27       "ctor": GameplayStrategy
28     }
29   ],
30   sceneMap: {
31     "MainScene": MainSceneBlueprint,
32     "IntroScene": IntroSceneBlueprint,
33     "OutroScene": OutroSceneBlueprint
34   }
35 }
36
```

Figure 20. Game Blueprint

In the figure above, we create the game blueprint that gets build in the main script. The game blueprint includes three scenes; intro, outro, and the main scene that has most of the game logic.

The scene map as well as the VFL, are arguments passed to the Game class that maps each scene to its blueprint with its layout. Moreover, we can see two strategies that are not scene related, they are always active regarding which scene is currently active. Hence, we can distribute our code based on the scope to which each file belongs as shown in figure 21.

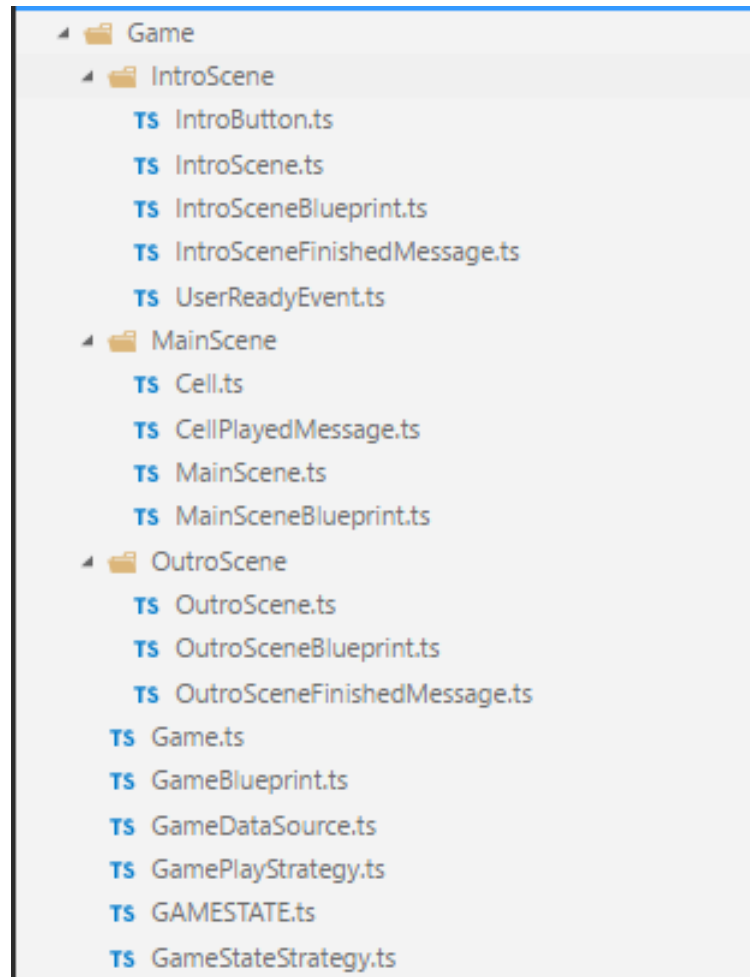


Figure 21. Folder Structure

By allowing developers to structure their folders based on the scope of the file, it will be easier to manage the files and to know what is each class responsible for.

3.1.3 Scene Blueprint

Every scene has its own blueprint that includes a tree of entities and game objects to be built. Similarly, the scene blueprint is itself nested inside the game blueprint. Figure 22 shows the blueprint for outro scene.

```
1 import { TilingSprite } from "../../WebGLadiator/src/GameObject/TilingSprite";
2 import { Text } from "../../WebGLadiator/src/GameObject/Text";
3 import { OutroScene } from './OutroScene'
4
5 export const OutroSceneBlueprint = {
6   "name": "OutroScene",
7   "ctor": OutroScene,
8   "lazy": true,
9   "vfl": [
10     "HV:|[TilingSprite]|",
11     "H:|~[Result(120)]~|",
12     "V:|~[Result(30)]~|"
13   ],
14   "blueprints": [
15     {
16       "name": "TilingSprite",
17       "ctor": TilingSprite,
18       "source": "Resources/bg_dark_blue.png"
19     },
20     {
21       "name": "Result",
22       "ctor": Text,
23       "text": "Result",
24       "options": {
25         fontFamily: 'Arial',
26         fontSize: 26,
27         fill: 0xffffffff,
28         align: 'center'
29       }
30     }
31   ],
32 }
```

Figure 22. Outro Scene Blueprint

In the figure above, text and animated sprite game objects are already implemented by WebGLadiator and reused by referencing them in the blueprint and passing their equivalent arguments e.g font style for texts. On the other hand, outro scene requires its own logic so a different class was created for it in a separate file.

3.1.4 Logic

Scenes have their own blueprint that will build the scene and layout it yet the logic for each class should be written inside. Considering the outro scene, it is responsible for displaying the winner after the game is finished and informs the game to go back to intro scene. figure 23 shows the code for outro scene. Moreover, if we consider a background image, it only requires using layout, hence no need to create a separate class for it since in most of the cases it does not have any logic.

```
7 export class OutroScene extends Scene {
8
9   listenToHIDEvents() {
10     super.listenToHIDEvents(true);
11     this.registerHIDEvent('pointertap')
12       .first();//to avoid memoryleaks, take first tap only
13     .subscribe((value) => {
14       this.sendEvent(new OutroSceneFinishedMessage(this));
15     });
16   }
17
18
19   start() {
20     GameDataSource.getInstance().obsData
21       .map(data => data.state.winner == data.playerId ? "Winner" : data.state.draw?"Tie":"Loser")
22       .subscribe(value => {
23         this.getNode("Result").text = value;
24       })
25   }
26 }
27
```

Figure 23. Outro Scene Logic

In the figure above, we can see two functions; one which is responsible for HID events and the other is responsible starting the logic. The reason for that is to divide the responsibility between the functions which is a common approach in programming. outro scene waits for a user tap then sends a message to the game to switch back to intro scene, while start method reads the current result to display the text based on a win or a loss. We can see here that the life cycle is completely hidden, the constructor is not needed, methods to destroy the scene are not added since WebGLadiator does that without intervention from the developer.

As per the methods, all methods get called by the builder without intervention from the developer and it is up to him whether to include them or not, we use three methods;

- 1) listenToBusEvents: used to listen to messages.
- 2) listenToHIDEvents: used to listen to HID events.
- 3) start: generic and can be used to write both of the functions above, this is can as well be used to run the code internally.

If a developer has a class that only needs to listen to messages, then he can add the first message and does not need to add the other ones. Similarly, if he wants to use a class

that only needs to listen to HID events e.g button, then he can use the second method alone. On the other hand, the last method can be added to include listen to any event or also run internal code like in Unity.

3.1.5 Scene Management

In order to manage the game state, which scene the game should be, we will a game state strategy the global class that will be responsible to load the game and as well to switch between scenes. Figure 24 shows the code in start function of the strategy.

```
8  export class GameStateStrategy extends Strategy{
9      start(){
10         let self = this;
11         System.getInstance().getSystem("resource").preload("Resources/resources.json").subscribe({
12             next:function(value){ //ReactiveX: stream emitted a value
13                 console.info(value);
14             },
15             complete:function(){ //ReactiveX: stream is completed, no more emissions
16                 document.getElementById("container").style.display = "none";
17                 SceneManager.getInstance().loadScene("IntroScene");
18             }
19         });
20     };
21
22     GameStateSource.getInstance().getStream().subscribe(function(data){
23         if(data && data.state && data.state.currentTurn !== null){
24             if(data.state.winner !==null || data.state.draw == true)
25             {
26                 SceneManager.getInstance().switchScenesTo("OutroScene", true);
27                 return;
28             }
29             if(SceneManager.getInstance()._activeScene.name == "MainScene")
30                 return;
31             SceneManager.getInstance().switchScenesTo("MainScene", true);
32         }
33     });
34 }
35 listenToBusEvents() { ...
43 }
44 }
```

Figure 24. Game State Strategy

In the figure above the class extends Strategy class and in the start function, which gets called automatically by the builder, we load the resources and for now we just log the progress until it is completed then we load the intro scene. In the second part of the function, we listen to the changes in the game data source and based on the current state we transition to main or outro scene. It is worth mentioning here, that the code is as per server implementation and the state is used as per the server works.

3.1.6 Playing The Game

The result of our code will run on different devices with different screen sizes. By using the layout system via VFL our game will run on any screen size regardless of the aspect ratio. Figure 25 shows a screen shot of two opponents playing against each other where every player has "X" as his mark.

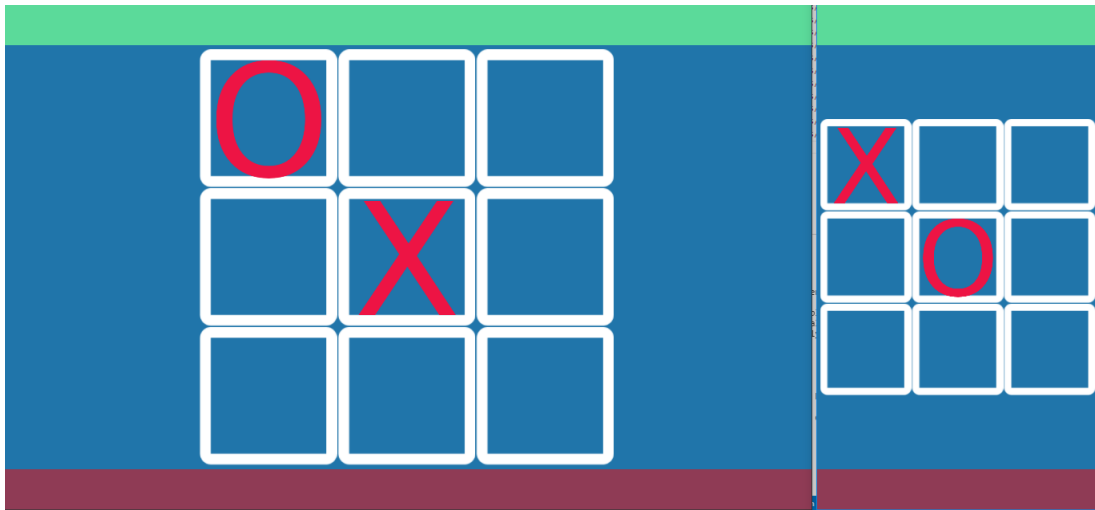


Figure 25. Responsive Tic Tac Toe

In the figure above, we see that the game fits any screen regardless of the aspect ratio and orientation. The header and footer are usually absent in Tic Tac Toe games, but we added them as a show case, they both are tiled sprites with fixed height and full width. On the other hand, the board takes all available space without stretching, we achieved that by enforcing the inner aspect ratio of the board.

During the game play, both intro and outro scenes will send messages to this strategy to inform the strategy to change the scene. Figure 26 shows the code in event function of the strategy.

```
8  export class GameStateStrategy extends Strategy{
9      start() {
34  }
35      listenToBusEvents() {
36          super.listenToBusEvents();
37          this.registerEvent(IntroSceneFinishedMessage).subscribe(message=>{
38              SceneManager.getInstance().switchScenesTo("MainScene", true);
39          })
40          this.registerEvent(OutroSceneFinishedMessage).subscribe(message=>{
41              SceneManager.getInstance().switchScenesTo("IntroScene", true);
42          })
43      }
44  }
```

Figure 26. Game State Events

In the above figure, we can see that events function which also gets called automatically is separate from the start function as every function has a different responsibility. Accordingly, the code here is responsible for listening to finishing messages and switches to the next scene.

3.2 Web Tailored Engine

From a web developer perspective, the ability to change from one field to another is a matter of adapting. Presenting the engine the same way it is for the web will help developers adapt quickly. Adding to that, the similarity web developers would see can motivate them to start with game development as a way to learn new skills the easy way.

3.2.1 Web Technologies

As per the web standards and technologies, WebGLadiator retained the structure as is in order to reflect it for game development. The approach to game development is similar to that of web development in terms of the three technologies: JavaScript for code, HTML for structure, and CSS for design and layout that were equivalently replaced TypeScript, JSON, and VFL. Moreover, the code organization and the separation of concern is similar to what modern web framework provides, e.g Angular. Let us have a look at a code example from angular in Figure 27

```
@Component({selector: 'my-cmp', template: '...'})
class MyComponent implements OnInit {
  ngOnInit() {
    // ...
  }
}
```

Figure 27. Angular Code Example

The reason why we give this example from angular is that angular is a popular framework and other similar frameworks use the same convention. If we look at MyComponent the component decorator '@Component' takes the template argument which is HTML snippet that in our case it is called blueprint. The reason why we do not use 'template' as a wording is that template has many meanings and might confuse developer. We can as well see the ngOnInit function which the function called upon initialization however without being called from the constructor in the same way our builder calls the functions. Figure 28 shows our code example.

```

4  import { UserReadyEvent } from './UserReadyEvent';
5
6  export class IntroScene extends Scene{
7      listenToHIDEvents() {
8          super.listenToHIDEvents(true);
9          this.registerHIDEvent('pointertap')
10             .first();//to avoid memoryleaks
11             .subscribe((value) => {
12
13                 this.sendEvent(new UserReadyEvent(this));
14                 this.getNode("Tap").visible=false;//pend
15                 this.getNode("Pending").visible = true;//wait for opponent
16             });
17      }
18  }
19
20 }

```

Figure 28. Intro Scene Code Example

In the code example, we see how every function will be automatically called by the builder without having to call them from the constructor which is absent this would eliminate the confusion for developers as per what to name functions and where to call them. This way it looks similar to web frameworks and as well reduces the overheads when it comes to inheriting from classes with a constructor. Moreover, developers will be using same naming conventions for same functions. Adding to that, creating the scene, was done by extending the scene class and adding it to the game was done through the blueprints.

3.3 Agile Development

Agile software development requires that starting and ending development of the software should be as smooth and fast as possible while being ready to make any changes at any time. If we apply that on our case, then starting with the new game is a matter of creating a blueprint. Afterward, adding new code is simplified by life cycle management and builder, the developer will start writing his code directly. This means that a developer will only concentrate exclusively on the writing the code for his logic rather than the UI. Throughout the development process, if needed, a developer can just remove unnecessary entities or game objects from the equivalent blueprint and if a slight change, like changing a parameter, is needed then it could be done via blueprint without touching the code.

3.3.1 Easy Game Setup

Starting from a scratch with a new game can be frustrating as the developer has to make a lot of decision that might or might not be correct. If we consider only creating the layout of a game, then using VFL will avoid having to write our own calculations and

aspect ration consideration. However, in our case, a developer would just need to create a blueprint file and add to it the VFL required and that is very similar to the design steps needed for web development with CSS. Adding to that, since we have reusable elements, a developer can simply use our tiled sprite as a UI debugging technique to visualize his layout before starting.

Now that the developer knows how his game is going to look, he will start by using proper controls he needs for the game that we have already made. Upon adding menus, buttons and what he needs for his game the user can judge the interaction inside his game. Followed by that, the user then can easily create his own code and add it accordingly without having to worry about his previous actions because it is just a matter of removing them from the blueprint file.

3.4 Organized Work flow

With the use of software design patterns and the well-organized engine, the developer will concentrate on simplicity rather than hard work. Developers will concentrate on completing their game without the need to worry about the engine, they will build the game until the end without having to write unnecessary code that was taken care by the engine e.g. One of the pitfalls of large LOC is that it is hard to remember the intent for writing them as well as hard to change them on demand. Figure 29 indicates how coding was done before and after WebGLLadiator.

```

1 //https://pixijs.github.io/examples/#/basics/container.js
2
3 var app = new PIXI.Application(800, 600, {backgroundColor : 0x1099bb});
4 document.body.appendChild(app.view);
5
6 var container = new PIXI.Container();
7
8 app.stage.addChild(container);
9
10 var texture = PIXI.Texture.fromImage('required/assets/basics/bunny.png');
11
12 // Create a 5x5 grid of bunnies
13 for (var i = 0; i < 25; i++) {
14     var bunny = new PIXI.Sprite(texture);
15     bunny.anchor.set(0.5);
16     bunny.x = (i % 5) * 40;
17     bunny.y = Math.floor(i / 5) * 40;
18     container.addChild(bunny);
19 }
20
21 // Center on the screen
22 container.x = (app.renderer.width - container.width) / 2;
23 container.y = (app.renderer.height - container.height) / 2;
24
25
26 export const IntroSceneBlueprint = {
27     "name": "IntroScene",
28     "lazy": true,
29     "ctor": IntroScene,
30     "manifest": "Resources/MainSceneManifest.json", //this is okay for now since we are graceful
31     "vfl": [
32         "Hv:|[TilingSprite]|",
33         "Hv:|[Tap(120)]|",
34         "Vv:|[Tap(30)]|",
35         "Hv:|[Pending(120)]|",
36         "Vv:|[Pending(30)]|",
37         "Hv:|[Button(400)]|",
38     ],
39     "blueprints": [
40         {
41             "name": "TilingSprite",
42             "ctor": TilingSprite,
43             "source": "Resources/bg_dark_blue.png"
44         },
45         {
46             "name": "Button",
47             "ctor": MockButton,
48             "vfl": [
49                 "Hv:|[TilingSprite]|",
50             ],
51             "blueprints": [
52                 {
53                     "name": "TilingSprite",
54                     "ctor": TilingSprite,
55                     "source": "Resources/red.png"
56                 }
57             ]
58         }
59     ]
60 },
61

```

Figure 29. Before and After WebGLLadiator

3.4.1 Coherent Engine

The purpose of creating an engine instead of using one is to be in control while making games. The structure and coherence are spread all over the engine from an entity to a

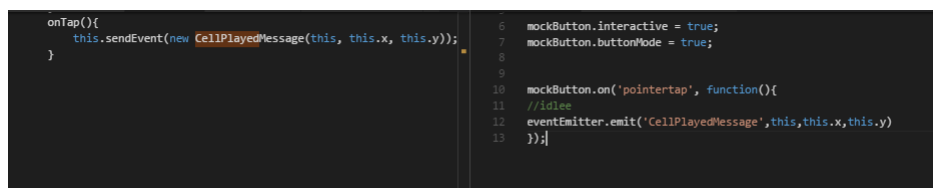
system. Not only will this mean that it is easier to develop games, but also to understand another developer's code and work in a similar pattern all over a team. This process is generally costly and requires

Using software design patterns along with a modified ECS translates into this easily understandable and usable engine. If we consider a developer who wants to know not only how to make games with our engine but also how was our engine made, then he can relate our decisions to our conventions used e.g he will google it. For example, if a developer sees a Facade and multiple Adapters, yet did not understand the naming, he can google about facade and adapter software design patterns and that might help him understand how our decisions were made.

3.4.2 Readability

To make our code readable we had to use conventional naming for our files that simplifies the understanding of what each file does. Classes are implemented based upon their functionality, for example using the builder, Singleton, and strategy instead of manager and utility as a name. along with separating the concern between visuals and logic whilst assisting developers, we made the engine simple enough to assist developers but no more simple. By reducing the complexity, readability in our case went beyond our own code to reach developers' code among each another.

When a developer wants to add his code, he doesn't have to worry about how the code should look like because that is managed by the engine, all the developer will concentrate on is to make the logic work and everything else will be taken care of the engine. For example, if life cycle management was missing it will mean that the developer will have to maintain the life cycle his own. Imagine having two developers, then everyone will write it his own resulting into two different implementations of the very same thing. Figure 30 shows between WebGLadiator code and usual code.



```
onTap(){
  this.sendEvent(new CellPlayedMessage(this, this.x, this.y));
}

mockButton.interactive = true;
mockButton.buttonMode = true;

mockButton.on('pointertap', function(){
  //idle
  eventEmitter.emit('CellPlayedMessage', this, this.x, this.y)
});
```

Figure 30. WebGLadiator Code Comparison

If we consider the example above, then we can see that we eliminated the need to call the method from the constructor while also we made sure all developers will use the same name for the method e.g some one can call initEvents or initHID. Moreover, in the second method, we are emitting the 'CellPlayedMesage' event, which is a wrong string

missing the second 's' letter. Hence, no one will receive this message and the developer might not be aware of it.

4 Discussion

The project could distinguish itself from other available WebGL game engines by bridging game development with web development. Moreover, we added around it responsive game development which facilitates creating games for any screen size and any aspect ratio that web developers are familiar with. Adding to that, being extensible and modular would eliminate the need to build an in-house engine by game companies. In this chapter, we are going to compare WebGLadiator with other available WebGL game engines.

While taking decisions, many of the decisions were different from those of other game engines. WebGLadiator takes its responsibility in creating games just like other engines. Moreover, our different approach has added different aspects that are slightly or completely absent in other game engines. Hence, we are offering auxiliaries along with the package that developers can benefit from our extra features and approach while developing their games on top of our engine.

4.1 Comparison

When comparing the currently available game engines we will be focusing upon source code availability, readability, learning curve, target platform, and continuity of development. Moreover, the comparison is going to take place among engines that are built around WebGL technology.

4.1.1 List of Engines

There is an adequate number of game engines for WebGL with different features and design. Since WebGL is the main perspective for the comparison, it is worth mentioning that with the current state of web technology, specifically WebGL support. Figure 31 shows vote result for most popular WebGL game engines provided by ClayIO a platform that publishes web games.

| Name | Cost | Popularity | Rating | Tags | Last Release | Details |
|-------------|------------|------------------------|--------|--|---------------|------------------------------|
| Construct 2 | varies | <div><div></div></div> | ★★★★☆ | game-maker free 2d 3d webgl sounds collisions physics | Aug 19th 2014 | More Details |
| Phaser | free (MIT) | <div><div></div></div> | ★★★★☆ | flash-like 2d sounds collisions physics typescript webgl free | Jul 11th 2016 | More Details |
| pixi.js | free (MIT) | <div><div></div></div> | ★★★★★ | 2d webgl free | Jun 12th 2016 | More Details |
| Three.js | free (MIT) | <div><div></div></div> | ★★★★★ | 3d webgl free | Jul 14th 2016 | More Details |
| PlayCanvas | free | <div><div></div></div> | ★★★★★ | 3d cloud-based free webgl sounds | Jul 22nd 2016 | More Details |
| Turbulenz | free (MIT) | <div><div></div></div> | ★★★★☆ | 2d 3d webgl sounds collisions physics debug networking | Dec 22nd 2015 | More Details |
| CAAT | free (MIT) | <div><div></div></div> | ★★★★☆ | 2d free webgl | Jul 2nd 2013 | More Details |
| enchant.js | free (MIT) | <div><div></div></div> | ★★★★☆ | 2d sounds collisions physics webgl free | Jan 4th 2016 | More Details |
| Panda.js | free (MIT) | <div><div></div></div> | ★★★★☆ | free 2d webgl mobile physics sounds modular | Feb 17th 2015 | More Details |
| Kiwi.js | free (MIT) | <div><div></div></div> | ★★★★☆ | 2d webgl physics free | Nov 15th 2015 | More Details |
| voxel.js | free (BSD) | <div><div></div></div> | ★★★★☆ | webgl 3d voxel sounds physics networking | Oct 4th 2015 | More Details |

Figure 31. Which HTML5 Game Engine is right for you? [Cla17]

The figure above lists HTML game engines by the cost of license where free means an open source project which we intend to do. Followed by popularity and rating equivalent to the number of downloads and votes, the higher value the better. Moreover, tags refer to what we call features and represents what each engine provides. From the features included, WebGLadiator provides 2D, sounds, free, WebGL, TypeScript adding to that WebGLadiator is HTML-like instead of flash-like tag, provides blueprints instead of game-maker tag.

4.1.2 Comparison between available engines

While Construct 2 is on the top of the list, the project became out dated as the last release was in 2014. Construct 3 is currently in beta version, Construct 3 is very similar to PlayCanvas which is to be discussed as well, hence for clarity, we will separate version 2 and 3. Construct indeed proved itself by adding a GUI which goes beyond being extensible to ease of use. Our perspective from a GUI is a bit different, WebGLadiator will replace GUI with blueprint files, the idea of using blueprints is that JavaScript developers are moving towards GUI-less approaches. for example using Sublime Text Editor instead of IDEs, using gulp and NPM instead of automation tools, since in JavaScript ecosystem everything runs in the console, for example, compiling, building, releasing and running web apps is mainly done using CLI.

Phaser is a flash-like game engine, with the initial intent to be the engine to replace flash with WebGL. Phaser uses JavaScript to develop games the same way in Flash and is close to an ECS architecture. While Phaser supports plenty of features even social platform plugins, Phaser is written in JavaScript 5 (without TypeScript nor es6) which makes the code quantitative with plenty of nested namespaces. For example, loading an image would be `Phaser.Game.load.image()` while loading a JSON file would be `Phaser.Game.load.json()`, Although both of them are resources and decoding them should

be delegated to a middle-ware. On the other hand, WebGLLadiator is written in TypeScript to make the code more understandable and is utilized by Software Design Patterns, specifically builders, in order to reduce the LOCs that would be otherwise required. Figure 32 shows two benefits included in WebGLLadiator for resource management and IntelliSense with TypeScript that speeds up the process of coding applications by reducing typos and other common mistakes through auto completion popups when typing, querying parameters of functions, query hints related to syntax errors, etc.

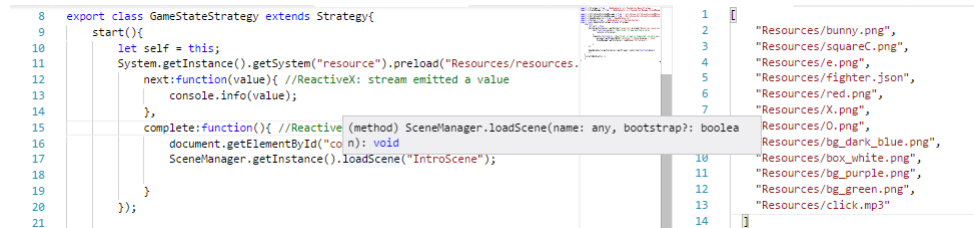


Figure 32. Resource Management and Intellisense

Pixi developers did the exact opposite of what Phaser did in terms of licensing, they made the software as a branding for themselves which lead to contracts with Kinder, Disney, Toyota and many other companies that Pixi developers (GoodBoyDigital) made benefit of and added to their gallery page under <http://www.pixijs.com/gallery>. As a result, it was a push for them to keep working for Pixi. The downsides of Pixi is that it is just a rendering engine that can and is mostly used by most game engines including Phaser. One of the contributing reasons why it is voted 5 out of 5 is that it gives freedom for companies to make their own engine on top of it as quoted by reviewers "Great as a blazing fast rendering base for bigger apps and games. I love that performance is a first class citizen here, with no wasteful or expensive default settings"[Dak16]. As per WebGLLadiator, we are using Pixi for graphics system to make WebGLLadiator a superset of PIXI that adds for the least sound and animation systems.

Another engine in the is PlayCanvas which is a cloud based IDE and game engine that follows the ECS approach. PlayCanvas is a full ECS engine that goes beyond development to deployment. One downside with PlayCancas is that it uses ES5 which involves writing a lot of code. Imagine inheritance over a prototyping language, and using it for a casual game makes it an over simplification that the LOCs required are just too much. Figure 33 shows lines required just for a Button.

```
var Button = pc.createScript('button');

Button.attributes.add('displacement', { type: 'number', default: 0.00390625 });
Button.attributes.add('event', { type: 'string' });

// Initialize code called once per entity
Button.prototype.initialize = function() {
    this.pressed = false;
    this.min = new pc.Vec3();
    this.max = new pc.Vec3();

    this.mousedownListener = function(e) {
        this.pressed = e.clientX < this.min.x || e.clientX > this.max.x;
        this.bind(this);
        this.mouseupListener = function(e) {
            this.release();
        }.bind(this);
        this.touchstartListener = function(e) {
            var touch = e.changedTouches[0];
            this.pressed(touch.clientX, touch.clientY);
        }.bind(this);
        this.touchendListener = function(e) {
            this.release();
        }.bind(this);

        window.addEventListener('mousedown', this.mousedownListener, false);
        window.addEventListener('mouseup', this.mouseupListener, false);
        window.addEventListener('touchstart', this.touchstartListener, false);
        window.addEventListener('touchend', this.touchendListener, false);

        this.on('enable', function () {
            window.addEventListener('mousedown', this.mousedownListener, false);
            window.addEventListener('mouseup', this.mouseupListener, false);
            window.addEventListener('touchstart', this.touchstartListener, false);
            window.addEventListener('touchend', this.touchendListener, false);
        });

        this.on('disable', function () {
            this.pressed = false;
            window.removeEventListener('mousedown', this.mousedownListener, false);
            window.removeEventListener('mouseup', this.mouseupListener, false);
            window.removeEventListener('touchstart', this.touchstartListener, false);
            window.removeEventListener('touchend', this.touchendListener, false);
        });
    };

    Button.prototype.checkForClick = function (x, y) {
        var app = this.app;
        var cameraEntity = app.root.findByName('Camera');
        var aabb = this.entity.model.meshInstances[0].aabb;
        cameraEntity.camera.worldToScreen(aabb.getMin(), this.min);
        cameraEntity.camera.worldToScreen(aabb.getMax(), this.max);
        if ((x <= this.min.x) || (x >= this.max.x) || (y <= this.min.y) || (y >= this.max.y)) {
            return true;
        }
        return false;
    };

    Button.prototype.press = function (x, y) {
        if (this.checkForClick(x, y)) {
            this.pressed = true;
            this.entity.translate(0, this.displacement, 0);
        }
    };

    Button.prototype.release = function () {
        var app = this.app;

        if (this.pressed) {
            this.pressed = false;
            this.entity.translate(0, this.displacement, 0);
            app.fire(this.event);
            app.fire('game:audio', 'Smooosh');
        }
    };

    // update code called every frame
    Button.prototype.update = function(dt) {
    };

    // swap method called for script hot-reloading
    // inherit your script state here
    Button.prototype.swap = function(old) {
    };

    // to learn more about script anatomy, please read:
    // http://developer.playcanvas.com/en/
```

Figure 33. Creating a Button with PlayCanvas

While we can see the plenty lines of code to create only one game object are okay, but we can still minimize that the way we do it in WebGLLadiator as in 34, we can also note that we don't need to dispose the events, managed automatically, and also we have support 'tap' event.

```

57   export class ConsoleButton extends Button {
58     onIdle() {
59       console.warn("Idle")
60     }
61
62     onTapped() {
63       alert("TAPPED")
64     }
65
66     onActive() {
67       console.info("ACTIVE")
68     }
69     onHover() {
70       console.log("HOVERED")
71     }
72
73   }

```

Figure 34. Button

In addition to the engines mentioned earlier, it is worth mentioning the coming back engine by Black Storm Labs called GameClosure. Although the project was almost closed for 2 years, by introducing Instant Games GameClosure came back to deliver EverWing, the biggest instant game. Noteworthy here, that while we use Autolayout, Facebook has their own Yoga.js layout library which could be used by GameClosure. Up to the list, GameClosure, without our additions like layout and lifecycle management, is the closest to WebGLadiator. However, The difference between WebGLadiator and GameClosure is we focus on web approach while GameClosure focuses on mobile approach from a UI designer perspective. Strictly, WebGLadiator focuses on bringing the game for web developers while being an ECS while GameClosure concentrates on mobile developers and the game UI rather than the game itself as in Figure 35.

```

var MainView = Class(View, function (supr) {
  this._def = {
    layout: 'linear',
    children: [
      {backgroundColor: 'red'},
      {backgroundColor: 'blue'}
    ]
  };
});

```

Figure 35. Creating UI with GameClosure

4.1.3 WebGLadiator

WebGLadiator, in terms of initial stage and vision, used available open source libraries which assisted in our journey and eliminated the need to write our own graphics, sounds, layout libraries. Instead, we just utilized them all together. In addition, WebGLadiator is targeted towards JavaScript developers and the way they write their code for the web. In other words, developers interested in WebGL will find it easy to learn because it is written for web developers to be used for web technologies.

In terms of a website development, developers usually write their code in JavaScript, styling, and lay outing in CSS, and visuals using HTML. For WebGLadiator, CSS is equivalent for VFL and HTML is equivalent for Blueprint. Even though we might have developers coming from Unity, specialized in CSharp, to develop a game for the web would require them to work with JavaScript and this can as well benefit from WebGLadiator.

Using WebGLadiator might be perceived as a risk, it is a safe step if considered specifically when developers prefer using plain old style. Many companies nowadays do the very same steps to obtain their version of WebGLadiator. The absence of developers who could actually implement the project makes it pretty difficult to control their engine which develops a cumbersome burden over time. WebGLadiator translates the technical details that would otherwise be difficult to understand by a new developer, into simple code that a developer can use until he is knowledgeable and then can improve the code if need be.

4.2 Decision Making

The major decision for this project was a challenge itself; should I make an engine from available libraries or should I use an engine and make it extensible? another decision was how to choose which library or technology for this project and specifically why WebGL. Adding to that, the use of software design patterns was a side decision that introduced new challenges and solutions.

4.2.1 Did I reinvent the Wheel?

Although WegGLadiator might look like a way for reinventing the wheel, it is more viable to say that it was more using the wheel to invent the car. The game engine part itself was a way of reinventing the wheel, but using available libraries it made least effort to obtain the result. The additions, blueprints, manifest, layout, and life cycle management were a completely new way of making an engine that made WebGLadiator recognizable among the others.

The addition of a chrome extension is by itself a new addition to the game engine, and a stress out that if we develop an engine for the web we have to use technologies

for the web. In other words, we did add new features that are otherwise absent in other engines. Moving slowly and carefully, while planning ahead made the engine extensible enough to use an extension.

4.3 Critics

4.3.1 Competing Other Engines

If we are to consider game engines by the architecture they use, for example, ECS in Unity. Then WebGLadiator then we have to consider the long lifetime of Unity and engines alike. If however, we consider the simplicity, then WebGLadiator simplifies the process of developing a game by addressing JavaScript developers with same development process under the hood. This means that we utilize and ECS in the way a simple game should be done.

4.3.2 Software Design Patterns for Games

Software Design Patterns are meant for any kind of software with complicated structures. The idea of using them is to organize the structure under a layer of the convention that would be otherwise easy to follow and understand. Another approach would be to call all classes with the names: Utility, Helper, Manager, Operator. The naming would work fine but it won't really give a meaning for the functionality that every class is responsible for.

4.3.3 Adoption Risk

The first question to come in mind when adopting a new software is the risk of having it acting unexpectedly or otherwise forcing the user, developer in our case, to follow a lot of constraints that would hold him back. WebGLadiator allows developers to do fall back to manually managed code as the builder is an assessment rather than an obligation. If the developer does not want to use a blueprint or use it partially then he is free to do it the way he wants and write his code just like he would do normally

5 Conclusion

Now that we know most of the aspects of the project we narrow it down to a conclusive state. The ecosystem for WebGLadiator is web friendly following the same approach as of web development yet for games. Web Developers, as well as WebGL developers, will find it easy to accommodate to the engine as it presents itself as a web first platform. Never the less, it makes use of Typescript as well which makes it easier to see code using IntelliSense and auto completion. In addition to that, the engine provides same convention JavaScript developers would see otherwise. For example, VFL to WebGLadiator is what CSS is to The Web, Blueprints are the same as HTML, and JavaScript remains used either way.

WebGLadiator can be used to eliminate the mandatory step when starting a new engine, which is to wrap it and extend it to be used as per the business requirements for a game company. WebGLadiator will take responsibility for this step in a conventional manner in order to be self-explanatory for web developers. Software Design Patterns used are easy to follow and understand how to use them since there are plenty of resources explaining them. Classes are named based on their functionality rather than generic terms, which is easier for software developers to understand what is every class responsibility.

Aside from the architectural assessment offered by WebGLadiator, WebGLadiator added features that are not focused upon in other engines, for example, responsive layout and blueprint. The usual step of having to start from a scratch with a game engine is taken care by WebGLadiator. Developers will not feel any more obliged to write their engine of scratch if however, they wanted to add they can inject their own code into the engine and at the before the game starts as in.

6 Future Work

Although the project is sufficient to make a game, it does not mean that it stops development at this stage. In reality, just like any other project, there is always something to add and make use of. A project does not stop at the same time it achieves something. As an extensible engine, it is expected the project will always be improved.

Modularity is a key point of the project, although it is modular there are still many other conventions that would make much modular. Dependency Injection and Inversion of Control are the first steps to a modular system. A GUI builder would make it much easier to use for a modular project. On the other hand, there are features to be considered as an enhancement for the project. Last but not least step by step the project should be open source for others to use.

6.1 Modularity

Modularity gives freedom to developers to include or otherwise exclude pieces of code on the fly without having to worry about removing or adding them manually. In our project, we handle that so far by importing only needed files. On the other hand, GUI game builder would make it a drag-drop functionality to add new components to the system which will simplify the workflow for games.

6.1.1 Dependency Injection And Inversion of Control

IoC is the general concept where control of flow is Inverted from client code to framework, which “Does something for the client”. SL (Service Locator) and DI (Dependency Injection) are two design patterns stem off from IoC.[Wiz17]. Using the following we will avoid having to import files manually. the snippet below is how angular handles the imports in 36

```
var app = angular.module('myapp', ['restangular']);
app.controller('UsersController', ['$scope', function ($scope, Restangular) {
  ...
}]
```

Figure 36. Angular

6.1.2 Game Builder

The game builder would allow a drag and drop functionality which is by itself superior. However, another major point of using it is to simplify the process of building blueprints and importing files. In the case of big games, blueprints will grow bigger and imports

will grow as well, the presence of a game builder would minimize the effort in import and managing them.

6.2 New Features

There are still some features not implemented yet but would make a big difference for the engine the most two important ones are using XAML in favor of JSON and making VFL Unobtrusive

6.2.1 XAML

JSON is meant to be used for data rather than the presentation of data. blueprints currently are in JSON but having them in XAML would make it more understandable to read the blueprint and in the future, game builder along with XAML would make a blueprint much easier and simpler as then it would look the same as HTML. Since javascript developers are not friends with GUI it should be kept in mind that the game builder will only be a middleware between the developer and the game hence a developer can make his game without the need of a game builder at all.

6.2.2 Unobtrusive VFL

VFL is a key point for lay outing games, right now we are using it directly in the blueprint by composing the VFL over the blueprint. However, if VFL was separated into its own layout file and then only referenced by graphical nodes we can share properties between different nodes same way for CSS classes.

6.3 Open Source

Open sourcing a project might not be a good idea just after it starts working, people might have the bad impression about the future of the project. That is why it is better to wait until the project is completely stable and then release publicly. Moreover, documentation is the only way for new comers to understand how would they use the project.

6.3.1 Non-Disclosure

Once the project is ready and stable it should be open sourced. Meanwhile, it is better to keep it private until the point where there is no risk in realizing it. The risk includes both developers not being able to use it after some time and the owner not being able to fix reported issues.

6.3.2 Documentation

Documentation is the only channel of communication between an owner and a user. Typescript alone, unlike ECMAScript, allows IntelliSense which is a vital part of documenting code. Adding to that, YUML could be used to generate diagrams small in size. Last but not least there are plenty of libraries that automatically generates a website out of comments, for example, Pixi is using it for their official documentation.

References

- [AS] Kathryn Huff Anthony Scopatz. Effective computation in physics. O'Reilly Media.
- [Bea] Vangie Beal. Tweening. <http://www.webopedia.com/TERM/T/tweening.html>.
- [Chi] Laureline Chiapello. Formalizing casual games: A study based on game designers' professional knowledge. DiGRA Conference. 2013.
- [Cla] Scott Clark. Web-based mobile apps of the future using html 5, css and javascript. <http://www.htmlgoodies.com/beyond/article.php/3893911/Web-based-Mobile-Apps-of-the-Future-Using-HTML-5-CSS-and-JavaScript.htm>.
- [Cla17] Clay. Which html5 game engine is right for you? <https://html5gameengine.com/tag/webgl>, 2017.
- [Con17] Josh Constine. Facebook messenger rolls out instant games worldwide. <https://techcrunch.com/2017/05/02/messenger-games/>, 2017.
- [Cuo] Jerry Cuomo. Anthony scopatz, kathryn huff. https://www.ibm.com/developerworks/community/blogs/gcuomo/entry/javascript_everywhere_and_the_three_amigos?lang=en.
- [Dak16] Dakota. Pixi review. <https://html5gameengine.com/details/13/pixi-js>, November 2016.
- [Dig] GoodBoy Digital. Pixijs. https://github.com/pixijs/pixi.js?utm_source=html5weekly#what-to-use-pixijs-for-and-when-to-use-it.
- [Gam] Reskin Games. What is reskinning? <http://www.reskingames.com/>.
- [Goo] Google. Extending devtools. <https://developer.chrome.com/extensions/devtools>.
- [Ker] Brian Kernighan. Software tools,. Addison-Wesley.
- [Mic] Microsoft. Introduction to reactivex. <http://reactivex.io/intro.html>.
- [Mol12] Willian Molinari. What is the game loop? <https://gamedevelopment.tutsplus.com/articles/gamedev-glossary-what-is-the-game-loop--gamedev-2469>, November 2012.

- [Mos] Hanspeter Mossenbock. Twin - a design pattern for modelling multiple inheritance. University of Linz, Institute for System Software.
- [Rut] Hein Rutjes. Autolayoutjs. <https://github.com/IjzerenHein/autolayout.js#getting-started>.
- [Shv] Alexander Shvets. Design patterns. https://sourcemaking.com/design_patterns.
- [War] Jeff Ward. What is a game engine. http://www.gamecareerguide.com/features/529/what_is_a_game.
- [Wik] WikiDots. What's an entity system? <http://entity-systems.wikidot.com/>.
- [Wiz17] Grid Wizard. Ioc vs di vs sl. <https://gridwizard.wordpress.com/2014/05/28/dependency-injection-vs-service-locator>, 2017.

Appendix

I. Glossary

Casual Games : Casual games can have any type of gameplay, and fit in any genre. They are typically distinguished by their simple rules and lack of commitment[Chi].

Instant Games : HTML5 cross-platform gaming experience, on Messenger and Facebook News Feed for both mobile and web. This new games experience allows people to easily discover, share, and play games without having to install new apps[Con17].

Tween : Short for in-between, the process of generating intermediate frames between two images to give the appearance that the first image evolves smoothly into the second image[Bea].

Blueprint : JSON format file used to structure the content of the game in a readable way to simplify the process of creating the scene graph via code.

Cascading Style Sheets (CSS) : Style sheet language used for describing the presentation of a document written in a markup language. Although most often used to set the visual style of web pages and user interfaces written in HTML and XHTML, the language can be applied to any XML document[Cla].

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Mohamad Qaddura**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
 - 1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
 - 1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

WebGladiator Game Engine For The Web

supervised by Margus Luik

2. I am aware of the fact that the author retains these rights.
3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 13.08.2017