AethelmE, HTML5 Game Engine with Multiple Canvas Elements

Kevin Gunawan
Computer Science Program
Bina Nusantara University
Jakarta, Indonesia

Raymond Bahana
Computer Science Program
Bina Nusantara University
Jakarta, Indonesia

***Abstract*— Game engine is software which ease the game development. As the processor power technology evolved and the HTML5 (HyperText Markup Language 5) specification are developed, browsers nowadays can natively (without any need for external plug-in) display animations and multimedia files (audio and video) using JavaScript as the programming language. Some of the features which are used in this research are HTML5's canvas and audio elements. The problem is that none of the existing free HTML5 game engines is able to support multiple canvas elements. This research will create a game engine, called AethelmE, which support multiple canvas elements as its unique feature. This game engine is also able to support sprite transformation, browsers compatibility, external asset loading, and audio format compatibility. This research successfully resulted in creating an HTML5 game engine which supports multiple canvas elements. It also supports all the scopes, with a small exception on sound format compatibility. Moreover, this research conducted a performance comparison testing of multiple HTML5 game engines, from which can be concluded that multiple canvas elements does not give significant performance gain compared to a single canvas.**

Keywords—Game engine, HTML5, multiple canvas elements

I. INTRODUCTION

Gaming industry has been an interesting and challenging field to work on, as it requires technical knowledge and creativity at the same time. It has a large audience, making the industry both easy (more people will play the game) and hard (people demand more features from the game) for the developer. This industry also has a wide range of technology on which games can be implemented and played. There are many game platforms, console and non-console, that exist nowadays. The variety of the game platforms results in difficulties in developing games, since they have different frameworks and different language to work on. Thus, in order to develop a game without taking a long time, it is better for not doing it from scratch, but instead using an additional layer, which people usually call a game engine.

The usage of game engine is especially important when it comes to new technology, such as HTML5. HTML5 is a new standard for HTML. Its technology is based on HTML, CSS (Cascading Style Sheets), DOM (Document Object Model), and JavaScript [1]. Since developers might not be accustomed to it and the technology itself has a steep learning curve, game engine is necessary to help developers create games with high quality within a short period of time.

Several HTML5-based game engines exist, each with advantages and uniqueness. Some examples are Crafty, Impact, LimeJS, Cocos2D, EntityJS. Some use JavaScript as their controller, while others use CSS. Some are GUI (Graphical user interface)-based, while others acts only as an external library.

These game engines have something in common: they generally use HTML5's new canvas element, and they use only one canvas. According to Boris Smus [2], using layered canvas will yield a better performance as opposed to using a single canvas. From this research, none of the game engines is using multiple canvas elements as their feature. Therefore, this research proposes a new approach in creating a game engine which will add support to multiple canvas elements feature.

II. THEORETICAL FOUNDATION

A. Game Engine

A game engine, by definition, is a core set of technologies combined into a single software package to accelerate game development [3]. Game engines let game developers to have more time to develop the game components rather than the technical matter, as it provides the technical abstraction. Game engines can be used in a form of included libraries or a stand-alone editor application.

An example of a stand-alone game engine which has its own editor is the Unity game engine. Unity is able to create both 2D and 3D games and has wide access to many format types of assets as it support models with the .3ds (3DS Max), .fbx (Autodesk), and .blend (Blender) extensions, making games created with Unity even quicker because developers do not have much to deal with model file formats [4].

B. Physic Collision Detection

Collision detection is one of the most basic requirements in game development, especially in computed physic environment [5]. It checks if an object, 2D or 3D, is touching or overlapping with another one or more objects. This is important because when the game is simulating real life's physics, objects that are bumping with each other are definitely respond with a collision reaction, whether it is moving in opposite direction or exploding. Therefore, collision checking is a basic feature for game developing.

The importance of collision detection in game development does not stop there. Even for game which runs at the minimum of 30 frames per second, the collision detection also has to be run thirty times per second in order to create a real time animation. And so,

This work is licensed under a Creative Commons Attribution 4.0 International License

an efficient and accurate algorithm for collision detection is needed for developers so processor power is not used up for detection collision only.

There exist several collision detection algorithms, each with their advantages and weaknesses. Note that this research will only limit its discussion on two-dimensional collision detection algorithms. Moreover, it will only discuss algorithms which are used inside the research‘s game engine.

The easiest algorithm to detect collision is by using radius. This algorithm treats all tested object as a circle. The algorithm basically checks the objects’ position and calculate the distance between the two. The algorithm then compare the distance with the combined objects’ collision radius. A collision happened when the distance is equal or less than the combined radius.

Another type of collision detection algorithm is detection by using the bounding box. Two object is colliding when their bounding box are overlapping. The idea of this algorithm is that calculating collision using box-shaped is easier than using complex shapes like polygons or stars. This algorithm is divided further into two parts: the Axis-Align Bounding Box (AABB) and the Oriented Bounding Box (OBB) [6,7]. These two algorithms are similar in term of bounding box calculation, but works mathematically different.

C. <canvas> element

<canvas> element is a new features of HTML5. It enables web browsers to dynamically draw 2D images with procedural method [8]. JavaScript is used to script the generated graphic (i.e. drawing a rectangle or sphere, or draw a bitmap image). Apple was the first who introduced the canvas functionality, but the use is limited only for OSX‘s WebKit. Later, other browsers like Gecko browsers and Opera also implement it on theirs. Not long after that, WHATWG (Web Hypertext Application Technology Working Group) make the canvas as a standard for HTML5 [8]. Since <canvas> implements the rasterized procedural method, when an update happened inside the canvas (e.g. drawing the next animation frame), the whole canvas has to be cleared and the new shapes and images are re-drawn. This differs significantly compared to drawing with SVG (Scalable Vector Graphic), another new API (Application program interface) for drawing in HTML5. SVGs are vector- based image, therefore it consumes less memory and the raw data can be saved inside the DOM. When an update happened, only the data has to be changed and the browser will generate the new images automatically.

D. <audio> Element

<audio> element enables web browsers to play music files, synthesize sound, and generate/process speech natively without any need for external libraries and dependencies [9]. Previously, when a website‘s owner want to put audio (or video) embedded inside their site, they have to use external plug-in such as Adobe Flash in order for their multimedia files works on client‘s web browser. With HTML5‘s <audio> element, browsers can natively play music and songs, provided that the browser support the sound‘s file format.

III. Problem Analysis

Developing a game requires a lot of resources and preparation. In order to make one, developers have to plan the gameplay, the design, the characters. Also, they have to choose the suitable technology with the targeted game platform and market. It is going to take longer time if they still need to learn about new technology (in case they develop the game in new platform) and even longer if they code the game on low level programming layer. More time will be allocated for debugging and testing purposes.

In order to deal with those problems, developers commonly use game engines for creating games. It enables them to create games with shorter time, because it gives an additional programming layer of technology abstraction. This additional layer provides ease for the developers because they do not have to code in low level programming language. Some basic functionalities which had to be made manually by developers also sometimes provided by game engines, such as collision detection, physic engine, embedded audio engine, and several more.

The mentioned problems above become more visible for the HTML5 technology because of two things. First, by the time this paper is written, HTML5 is still an evolving technology. The HTML5 specification is still a draft and maintained daily by W3C. Therefore, it‘s hard to create an HTML5 website that‘s fully compatible with browsers that support old version of HTML5 [10]. Second, browsers have their own implementation of HTML5, mainly because of the evolving specification. This creates a separated website implementation for different browsers. An example of this is the different web audio API developed by Google and Mozilla. For those reasons, game engines have been a feasible solution for creating games in shorter time, since developers do not have to worry about the steep learning curve and they can use the given additional functionalities. Developers can choose from a wide range of choices of game engines, differs on what platform they support, on how difficult to use the engine, on what features can the engine support, and on what kind of game that the engine can produce.

A. Existing Solutions

As for HTML5 game engines, there are already a lot of them exist in the market, starting from the free engines to the paid ones. Each of them has their own features, advantage, difficulties, and drawbacks. There are some engines which emphasize on their lightweight size, such as lycheeJS and Crafty, while others emphasize on the performance, such as Playcraft Engine and Pulse [2]. There is also HTML5 engines that already uses GUI for the editor, such as Construct 2.

However, as far as the author‘s research, none of these engine support multiple canvas elements, even though using multiple canvas element have several advantages. According to an article in.html5rocks.com, <canvas> element‘s performance can be improved by utilizing several canvases, overlaid on each other [3]. Another advantage is developers can specify canvases‘ position according to their needs and favor

The existing solutions which this paper will describe are limited to free HTML5 game engines. Game engines that are used to create specific game genres (RPG, isometric, classic-style) are also excluded.

A.1. Crafty

Crafty is a free HTML game engine that utilize <canvas> element and/or DOM to render the entities‘ graphic. Similar to how JavaScript handles event, Crafty also use event binding to update the entities. Furthermore, it support custom events that can be triggered using the function Crafty.trigger() that will announce the custom event and trigger all entities that has been bound with that specific event. The uniqueness of this engine is that it does not use the usual inheritance concept for the entities. Instead, it uses the multiple inheritance or trait concepts [11]. To put it roughly, each

This work is licensed under a Creative Commons Attribution 4.0 International License

object in Crafty can get specific traits, depending on what components given to the object on initialization. So the code:

```
Crafty.e("2D, DOM, Text")
```

creates an entity that have the 2D, DOM, and Text components. Programmers can also create custom components using the Crafty.c() function.

In order for the engine to recognize the drawing canvas, programmers can do it two ways. First, by letting the engine to create the canvas element, automatically by using the function Crafty.init(). Second, by specifying an existing canvas with an ID #cr-stage using the function Crafty.viewport.init(). The major disadvantage of this engine is that it requires the programmers to understand significant knowledge of JavaScript (about the event binding and other things) before they can use all of this engine‘s features

A.2. lycheeJS

lycheeJS is another HTML5 game engine, made by Christoph Martens. It has a systematic folder structure for the game assets and resources. This is required because lycheeJS uses the folder position to determine the namespace and class name. The game engine package also comes with a game template that‘s ready to be edited, including the index.html file. lycheeJS also provide an interesting feature for exporting the game. Using the lycheeJS-ADK (App Development Kit), a toolchain for this game engine, developers can export their game to applications that can be natively ran on Linux Ubuntu/Fedora platform. Previously, the ADK also support Windows and OSX platforms, but it‘s outdated because of incompatible libraries and misconfigured gcc compiler [12].

The main drawback of lycheeJS is the steep learning curve. This engine requires the programmers to understand the concept of prototype in Javascript. Another drawback is that it has little extensibility, because all of the resources and folders are already provided. This also disables the engine for retrieving external resources (e.g. from other site).

A.3. gameQuery

gameQuery is an HTML game engine that uses jQuery as the base. The main feature of this engine is the DOM manipulation. So instead of using the <canvas> element, it uses DOM and CSS as the image manipulator and renderer. This inevitably broaden the browser‘s support compared to HTML5‘s canvas element, because DOM technology was invented before <canvas> element. Also, it supports tile mapping, box collision detection, and callback/function registration for periodic calls. This periodic callbacks is what updates the whole game. For example, to register the updating function, this function is used:

```
$.playground().registerCallback(function(){/
*Updating code*/})
```

gameQuery‘s weakness is that it‘s dependant to other Javascript library, which is jQuery, so the file size of the engine is sacrificed for rich features that jQuery can give.

A.4. Construct 2

Construct 2 is an HTML5 game engine made by Scirra. There are three versions of Construct 2 available for developers; free edition, which have a lot of limitation, but good enough to create a simple game, the personal edition, which is targeted to indie game developers, and the business edition, which is mainly used by for-profit game organizations.

Construct 2 has a distinguished quality compared to other HTML game engines: developers do not have to code to create games. Some coding logics still exists, but in a form of click-and-drag. Sprites and entities can be drag- and-dropped to the stage, called layout, while the game logic is put on event sheet. The basic component of display is called a sprite object, and behaviors can be added to supported objects, sprite included. This behavior objects can be defined to set common game functionalities for specific objects. An easy example of this is the physic behavior. Sprites that have the physic behavior will automatically move downward, as if it‘s affected by gravity, and others that don‘t implement it will not be affected.

Construct 2 also supports AJAX request, collision detection, multiple animations, path finding and AI (from behavior), and some webGL effects. Using the behavior-concept for objects, Construct 2 enables an wide open extensibility support for custom objects and behavior. This engine can also export the game to platforms other than web browsers, such as Windows 8, Windows Phone 8, iOS, and Android. Apart from the rich features of Construct 2, it still has several disadvantages. Since the editor is GUI-based, it has quite large installer, more than 100 MB. Developers also have to pay quite sum of money for creating game without limitation. The editor is also not cross platform.

A.5. Traffic Cone

Traffic Cone is a HTML5 game engine made by Joseph Mordetsky [13]. It also utilize two HTML5 <canvas> elements in order for the engine to work. It supports sprite animations, tile-based world support, tile mapping, intelligent draw routines, and basic support for isometric path finding, AI, and collision detection. Additionally, Traffic Cone can easily create isometric display using specified images. Using the statement:

```
new GameWorld(250, 250, 73, 73,
GAME_WORLD_STYLE_ISOMETRIC)
```

it will create a GameWorld object that defaults to isometric style of drawing. It has several constants that can be used to draw isometric world according to developer‘s need, for example GAME _WORLD_CELL _UNDERLAY for floor tiles that will be drawn under character sprites, and GAME _WORLD_ CELL_OVERLAY for pillar/wall which will be drawn over the character sprites.

Another feature of Traffic Cone is the composite sprite. Composite sprite is “a sprite that instead of being made of a single sprite sheet is instead made of multiple images”. It is useful when the game has a collection system, where different collections will be displayed differently on the stage. It comes with two types of composite sprites, the simple and complex one. The simple composite sprite means that the sub-sprite positions are all the same, so the engine doesn‘t have to deal with image‘s offset. While the complex composite sprite takes offsets, width, height of the image for determining the frames of the sub-sprite.

The drawback of Traffic Cone engine is that there‘s not much resources to learn how to use this engine. Not even the official website has documentation for this engine, although there are some examples that can be followed.

This work is licensed under a Creative Commons Attribution 4.0 International License

B. Proposed Solution

Judging from the existing game engines, only Traffic Cone that uses two <canvas> elements, while other engines only use one or none (but instead use DOM). When used carefully, multiple canvas could create a significant performance gain compared to using only a single <canvas> [14]. The concept is similarly used in Traffic Cone, when there are more than one <canvas>, the engine can determine which <canvas> should be updated and which one does not. This technique can be expanded more, by determining which part of the canvas that should be cleared before being redrawn.

When using a single <canvas>, the engine definitely have to clear the whole canvas before redrawing it. Although the performance cost is quite small, this cost will be accumulated quickly because typically, images is redrawn 30-60 times per second. This will impact more on computers with low-end hardware. Therefore, managing which <canvas> and specifying part that should be cleared will definitely give a much better performance.

This research will propose a solution for the HTML game engine problem, a new game engine called AethelmE, which support multiple <canvas> elements. This engine will be developed using JavaScript programming language.

IV. Design And Development

Figure 1 shows the main classes that forms the game engine. The main class of the engine is the AethelmE class. In order to minimize the chance of mistyping the engine's name, a shorthand class is also provided, which can be accessed by using the AE class. With this shorthand class, developers can create a new instance of AethelmE by using either new AethelmE() or new AE().

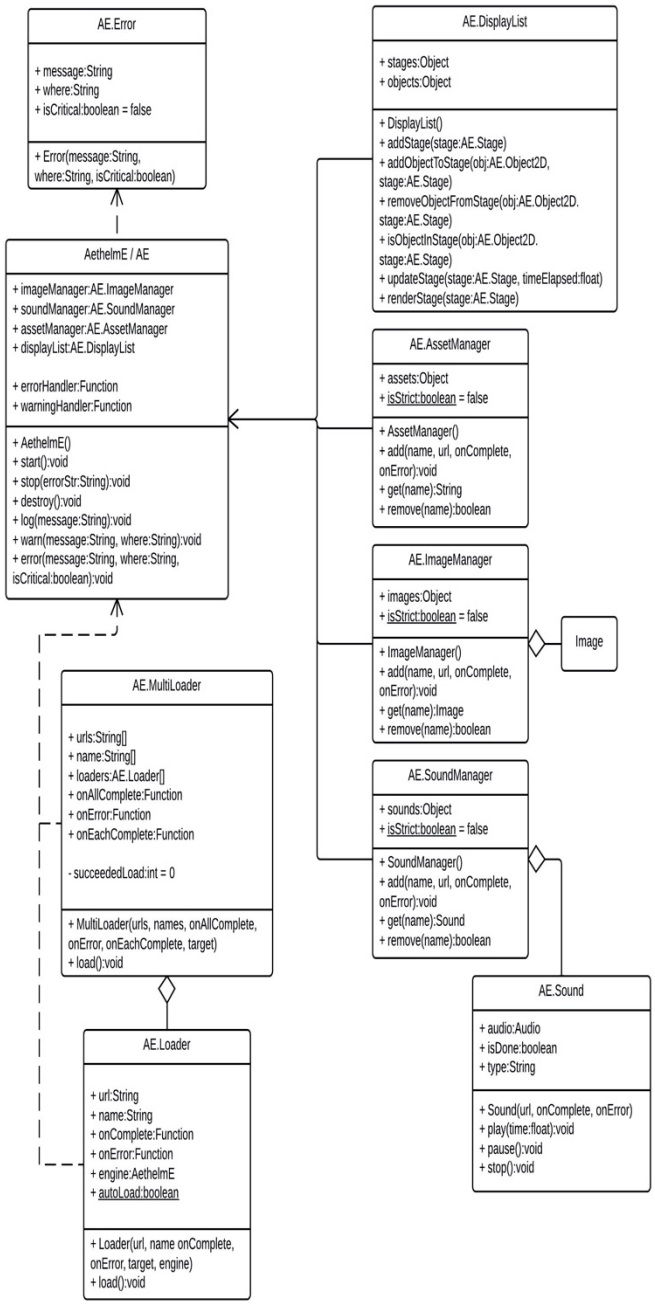

Fig. 1 Main Class Diagram

Note that all custom engine classes are using the word AE in front of the class name. This is done to prevent the class name clash with Javascript's reserved words, existing class name, and global variable names. This also applies to other engine's class name to maintain consistency and also enable the engine to be developed further without having to worry about global namespace.

This work is licensed under a Creative Commons Attribution 4.0 International License

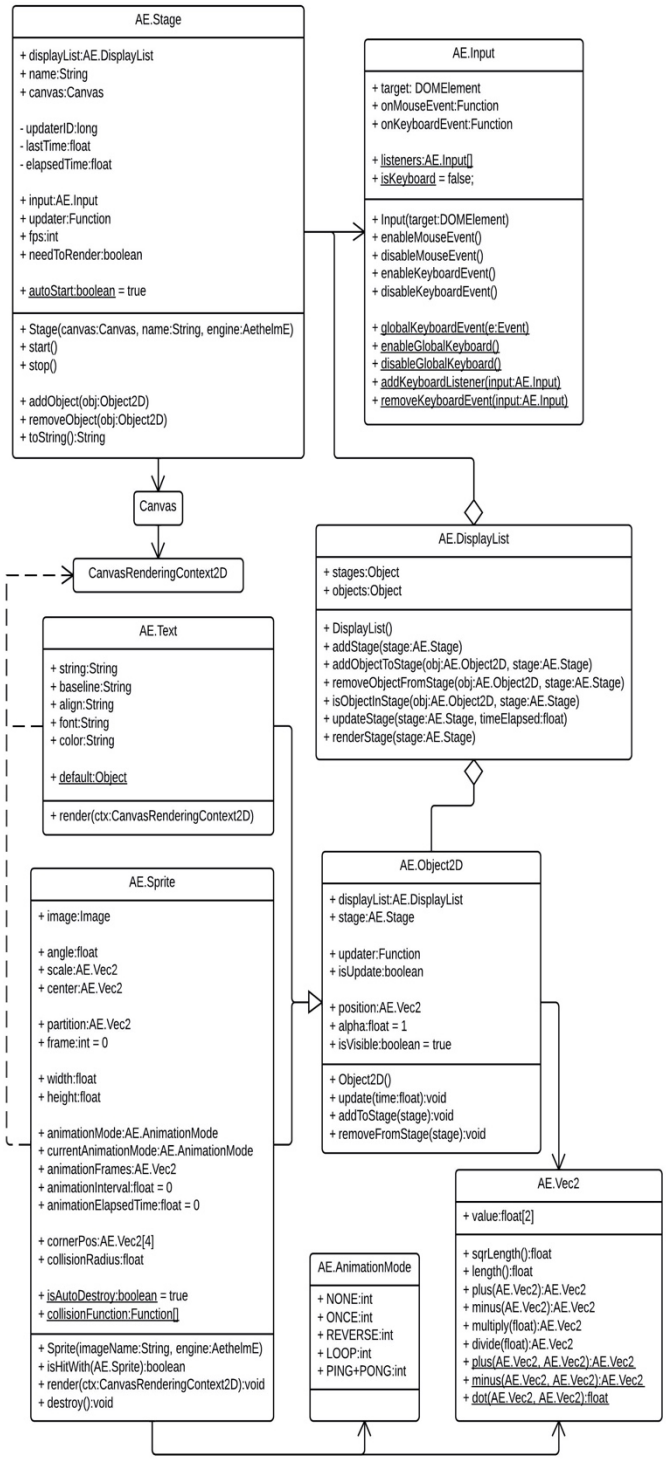

Fig. 2 DisplayList Class Diagram

Figure 2 shows the relationship between classes that handle rendering, with AE.DisplayList class as the center. As seen in the diagram, AE.DisplayList aggregates and stores two classes, which are AE.Stage and AE.Object2D. Also, this class does not only aggregate, but also pairs them so that each 2D object know on which <canvas> element it should render to.

A. AethelmE Class

This class is the main class of the game engine. Each instance this class saves all resource managers. It also saves a single instance of AE.DisplayList. There are also two functions which deal with error happening in engine. These two functions are the error and warning handlers. When the function AethelmE.error() is called, then the errorHandler function will also be called, with an instance of AE.Error is passed as the argument. Accordingly, when the function AethelmE.warn() is called, then the warningHandler function will be called with an instance of AE.Error is passed as the argument. AethelmE provides a default behavior for both errorHandler and warningHandler. Figure 3 And 4 show this behavior.

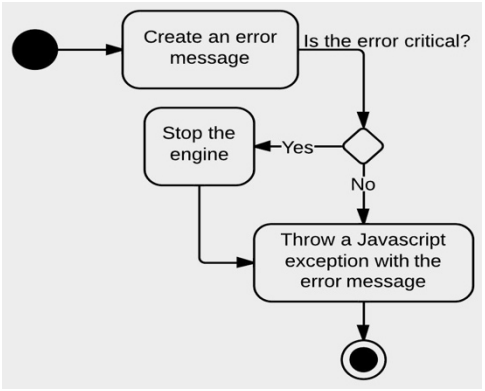

Fig. 3 errorHandler Default Behavior Activity Diagram

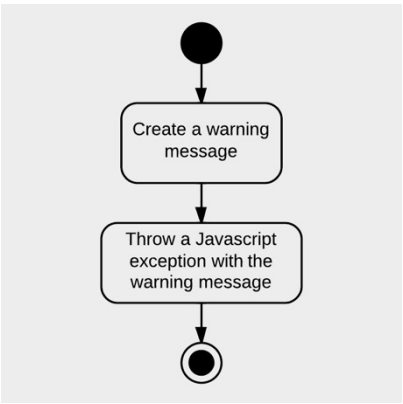

Fig. 4 waningHandler Default Behavior Activity Diagram

This class also provide the error() and warn() function, which developers can use to generate an error in-game. As a matter of fact, this two functions are only a function wrapper which create an instance of AE.Error, and then passed it to the corresponding handler. There are also log() function which can prints out message, as defined by logHandler function.

B. AE.Error Class

This class is a wrapper class which contains information when an error happened inside the engine. This error includes invalid argument type, underflow or overflow value, void operations, and accessing invalid asset. This class is mainly used inside AethelmE class, especially for the warningHandler and errorHandler functions, in which this class is passed as argument.

This work is licensed under a Creative Commons Attribution 4.0 International License

C. AE.Loader Class

The class AE.Loader is a utility class which gives ease for developers to download external resources, such as images, sounds, and text files. When this class first initialized, it is passed several arguments at the same time. First is the URL of the resource. The next argument is the asset‘s name. This name will be used to differentiate assets with another. The next two arguments are functions, which will be called when something happened with the resource loading. As the name implies, onComplete function will be called when the asset is fully downloaded, while onError function will be called when there‘s an error while the data is downloading. The next argument, target, is pretty tricky, as it deals with how function in Javascript works. target is the object that is bound to both onComplete and onError function.

The next argument is the engine, which determine which AethelmE engine the asset will be stored to. If it is not specified, the first created engine will be used. This argument is especially important because there can be multiple AethelmE engines exist on a single page (although it is not necessary), so AE.Loader has to know on which engine the downloaded data will be stored. Also, there‘s a single method exist in this class, which is the load() function. When this function is called, AE.Loader will start loading the resource. This step can be skipped if the static attribute autoLoad is true (which is the default value). While this attribute is true, the asset will be automatically loaded as soon as the AE.Loader object is initialized and instantiated.

D. AE.MultiLoader Class

The class AE.MultiLoader is a utility class, similar to AE.Loader, but is able to load multiple resources at once. AE.MultiLoader aggregates multiple AE.Loader objects to in order to load multiple assets. When an object of AE.MultiLoader is instantiated, it received several arguments at the same time and they are similar to AE.Loader‘s arguments. The first argument, urls, is an array that contains strings of the assets‘ URL. The second argument, names, is also an array that contains strings which will be paired to the asset‘s URL when it‘s done loading. The next three arguments, onAllComplete, onError, and onEachComplete, are functions which will be called when the matching event happened. The final argument, target, is the exact same as the target argument on AE.Loader; It will be bound on the three functions (onAllComplete, onError, and onEachComplete) to define the object that the keyword this will be referring to.

E. AE.ImageManager Class

The class AE.ImageManager stores images that are used inside the game engine. The class itself does not need any argument to be initialized. The image is stored when the function add() is called. This function requires several arguments to be executed. First is the name, which is a string that differentiates the stored images. The next argument is the url, which is a string that defines the image‘s address. The next two arguments, onComplete and onError, is the callback functions which will be called after the corresponding event happened. The way function load works is by using the existing event listener inside the Image class. It handles when the images is already downloaded and when there‘s error when loading it. Using this listener, AE.ImageManager can use the past functions from the argument to the appropriate event.

The downloaded image is stored on the images attribute. That it is not an array, but instead an empty Javascript object. The image is stored in a pair of key-value format, which uses the name argument as the key. A problem could arise when the name is already taken and paired with another Image. This is where the isStrict attribute is used. isStrict is a static attribute, which defines the behavior when there is a name clash. The default value is false, which will make the engine create a warning when there‘s a clash on the key.

F. AE.AssetManager Class

The class AE.AssetManager stores all other files which cannot be supported by AethelmE. This includes text files, JSON, and XML. Similar with other two manager classes, the assets attribute saves all assets in a key-value pair, with the name is stored as the key and the downloaded data is stored as the value. AE.AssetManager also has the static isStrict attribute, which defines whether it will create an error or a warning when there is a name clash inside the assets attribute. It also provide the add(), get(), and remove() function.

In order to download the data with unknown format, AE.AssetManager uses the Javascript‘s AJAX technology. AJAX uses the XMLHttpRequest class to download resources asynchronously. This data can be retrieved in a text format or binary format, depending on developer‘s need. For AE.AssetManager, the XMLHttpRequest uses only POST request, as it has larger data length limit compared to GET requests.

G. AE.SoundManager Class

This class stores any audio files that is downloaded and used by the engine. The class constructor has no argument, and it only initialize the sounds attribute. AE.SoundManager stores the audio file in a key-value pair inside the sounds attribute. When the add() function is called, it requires the name attribute, which it will be used for the key, and the downloaded file from the url attribute will be stored as the value.

The static attribute isStrict is also present in this class to prevent the engine to replace stored audio with the same name. When the value is false, the engine will create warning when a the name already existed, and will still replace the audio file. When the value is true, it will generate an error and depending on how the error handler work, it will or will not replace the audio file. AE.SoundManager class also provide function to retrieve the audio data using get() function, and to delete and free the memory for unneeded audio data using the remove() function. The remove() function returns a boolean value, indicating whether the deletion is successful or not.

H. AE.Sound Class

The AE.Sound class acts as a wrapper class for the HTML5‘s native Audio class. As seen from Figure 1, it has the audio attribute, which will save an instance of Audio class. The reason Audio class has to be bound inside another class is because it has no detection support whether the audio has completed the download completely. It can only detect streaming status, which can stop downloading before the data is fully downloaded. This is unfavorable for audio in games, because it will create choppy sounds when users are playing the game, especially those who have slow internet connection, and will definitely impact the gaming experience. In order to circumvent this problem, the AE.Sound class make use of the existing Audio‘s event handlers to detect whether the data has downloaded completely.

Figure 5 shows AE.Sound class‘ constructor. AE.Sound class uses two existing event handler from the native Audio class. They are the progress event and the canPlayThrough event. The progress event is called when the Audio is in the process of downloading, while the canPlayThrough is called when the browser thinks that

This work is licensed under a Creative Commons Attribution 4.0 International License

the downloaded data is long enough (but not yet complete) for the browser to play the audio. Using these two events, AE.Sound create a custom loading function which can detect when the audio has completely downloaded.

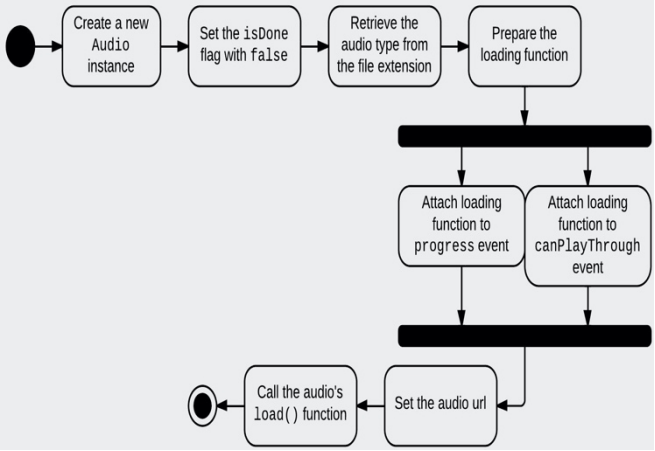

Fig. 5 AE.Sound constructor Activity Diagram

As seen on Figure 6, this loading function is dependent to the native Audio event. As the event continuously happening, the function will be executed again and again, until the audio is downloaded completely. In general, there are two main conditional statement inside the function. The first one checks whether the audio‘s metadata has already loaded. The Audio class loads the audio‘s information first before it actually stream the audio data. This information includes audio‘s title, duration, artist, album, genre, etc

The second statements checks if the streamed data already reached the end of the audio file, by comparing the end time of the streamed data with the total duration. If it has not reach the end, it will set the current time to the end of the streamed data. This is needed because Audio‘s implementation are different across browsers. The browser may continue the streaming even if the current time is still at point zero, but there are browsers which will stop the stream when the downloaded data is long enough for the user to hear. It will continue streaming when the current time is getting closer to the end of buffered data. So, in order to ensure all browsers behave the same, this function will actively set the current time to the end so all browsers will continue the streaming. If the data is fully downloaded, the current time will be reset to point zero, and the function will call the passed complete function. Then, the function will end its execution. AE.Sound class also support several functions for audio playback, using the play(), pause(), and stop() functions.

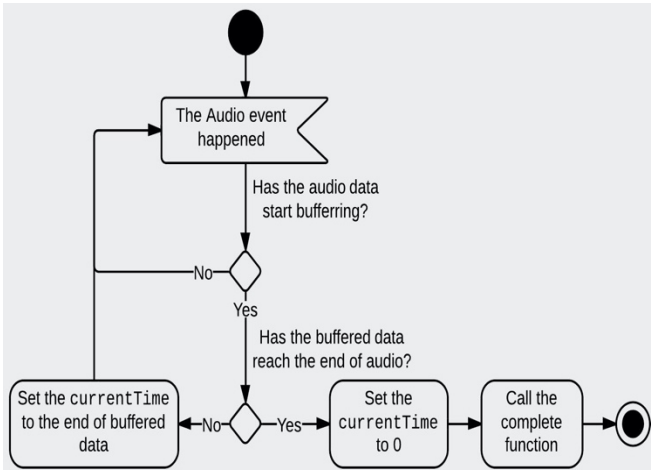

Fig. 6 AE.Sound onLoading function

I. AE.DisplayList Class

The AE.DisplayList class is the center class of sprites and stages, as it acts as the storage, manipulator, and rendering of all 2D objects in AethelmE engine. The class‘ constructor has no argument, because it only initialize its attributes, stages and objects. These two attributes stores all references to stages and objects that are initialized and used inside a single instance of AethelmE. As the names imply, the stages object stores all references to existing stages and canvas elements, while objects object stores all references to instantiated 2D objects. The way that these two objects store the references is unique, because both of them uses key-value pair scheme. Since JavaScript cannot use an object instance as an object‘s key, both of the objects uses the stage‘s name as the key.

The AE.DisplayList class also provides functions for manipulating stages and the objects inside it. Most of these functions are automatically called by other classes, so developers doesn't have to bother with AethelmE‘s object structure. The most basic one is the addStage() function, which will add the reference in the stages attribute and create an empty array in objects with the stage as the key.

J. AE.Vec2 Class

AE.Vec2 class is a utility class which saves two numeric values and can be manipulated as a two-dimensional vector.

This class only has a single attribute, which is the value array. It only saves two numeric values, with zero as the default values. This class is mainly used for AE.Object2D, AE.Text, and AE.Sprite class to define values that come in pairs. It is also frequently used for AE.Sprite‘s collision detection, with several vector calculation functions that helps with vector projection.

The AE.Vec2 class also provide several functions which is commonly used for vector calculations. The first function is the length() function, which returns the vector length. This class also provide the sqrLength() function which, as the name imply, return the squared length of the vector. Developer can (and encouraged to) use the sqrLength() function when the accurate vector length is not needed. For vector arithmetic calculations, this class also provides several functions with two variants, the method ones and the static ones.

This work is licensed under a Creative Commons Attribution 4.0 International License

K. AE.Stage Class

This is the class that contains all attributes and functions related to canvas element, updating, and rendering functions. While the one that actually renders is the AE.Object2D class and the one that call its render() function is the AE.DisplayList, AE.Stage is the one responsible of when to render the stage itself. Thus, AE.Stage is the one that calls the AE.DisplayList‘s renderStage() function. For rendering purposes, it has the private attribute updaterID, which saves the callback ID for the current rendering function call. It also has the lastTime and elapsedTime attribute.

As seen in Figure 7, AE.Stage constructor provides two ways to set the canvas attribute. The normal way is to pass the canvas element itself. For developer‘s ease, this argument also accept a string argument, which is the ID name of the canvas element, and the engine will automatically retrieve the element with the specified ID.

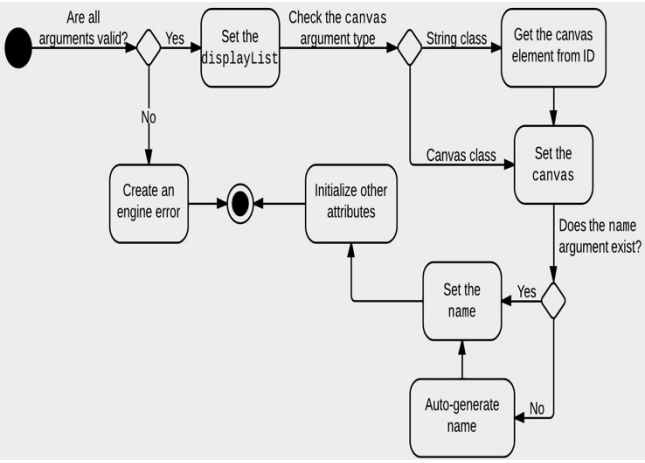

Fig. 7 AE.Stage Constructor Activity Diagram

AE.Stage also has the input attribute, which is an instance of AE.Input. For updating purposes, it also has the updater and needToRender attributes.The needToRender attribute is one of the most important feature of AE.Stage as this flag attribute is the one that determine whether the stage has to be re-rendered or not. This attribute has a default value of false, which means that the stage will not be rendered, and the last rendered image will stay as it is. This value will only change to true under certain conditions, such as a new object is added, an object is removed from the stage, or any of the object‘s visual attribute is updated. With the existence of this attribute, the stage can be rendered as needed so it will not consume as much processor resource as usual.

There‘s also a static Boolean attribute called autoStart which determine whether the stage is automatically started when an AE.Stage instance is created. Its default value is true. When it‘s set to false, all newly instantiated AE.Stage instances will not be started, and the start() function has to be called manually by developers. For manipulating the stage‘s workflow, this class provide two functions, the start() and stop() functions.

L. AE.Input Class

AE.Input class is a utility class that manage user input. Even if this class is created for AethelmE engine, it actually can support other HTML elements to a certain degree. This fact can be seen on AE.Input‘s constructor, as it requires oneargument, which is a DOMElement. As long as the specified element can receive user input, specifically mouse and keyboard input, the element can be attached with this class. Instances of AE.Stage will automatically have an instance of AE.Input inside it which is attached to the canvas element, so developers can use it automatically.

This class has several attributes, namely the target which save the attached DOM element, and both onMouseEvent and onKeyboardEvent which save the developer‘s custom callback function when mouse and keyboard events happened. This class uses Javascript‘s addEventListener() function to attach the callback functions to the elements. When the saved callback function in onMouseEvent and onKeyboardEvent is called, these functions will receive several arguments, depending on the event types. To save the processor power, these input events will not be active by default. In order for the element to receive user input, is has to be enabled using the enableMouseEvent() for mouse events and enableKeyboardEvent() for keyboard events.

M. AE.Object2D Class

AE.Object2D class is the parent class of all objects that are rendered and drawn on the stage. The two classes that inherit from AE.Object2D are AE.Text which define a text on the canvas and AE.Sprite which define a graphical render using images. Inside this class, its save the engine‘s display list reference in displayList and also the stage reference in the stage attribute. It saves the engine‘s display list because of the addToStage() and removeFromStage() functions are the wrapper functions of AE.DisplayList‘s addObjectToStage() and removeObjectFromStage() respectively. As such, this class will need to know the active display list reference. It also need to know the stage reference where the object will be rendered to, because it will need the canvas 2D context in order for the object is rendered, and the stage is the one who keep the canvas element reference.

N. AE.Text Class

AE.Text class is a children class that inherits from AE.Object2D. It define a text on a canvas. The written text will be the string from the string attribute. The align attribute is used to define the text. The font attribute is used for determining the text‘s font and style. Its type is string because it uses CSS‘s rule to determine the font. For example, to create a serif font with the size of 12 pixels, the font attribute will contain the string “12px serif”. It also applies to common styles, such as bold and italic. To create a bold text with previous values, the font attribute will contain the string “bold 12px serif”. The color attribute contain string that defines the text‘s color. It can contain the color’s name or can also be an RGB code in an rgb() function or hexadecimal value. The baseline attribute is useful for determining the vertical anchor point where AE.Object2D‘s position attribute points to. The common values for this attribute are top, middle, and bottom.

This class also provides the static attribute default, which consist of the default values of other attributes. When the class AE.Text is instantiated, it will initialize the attribute values with the ones inside the default object. This object is useful for developers who want to have a common font and style for all rendered text, without any need to change the attribute one by one. This class also have the render() function, which take a canvas context as the argument. When this function is called, it will render itself on the provided canvas context.

O. AE.Sprite Class

AE.Sprite is the basic class for displaying images on canvas. It inherits from AE.Object2D class. It uses the image that is saved inside AE.ImageManager. When AE.Sprite is instantiated, it requires an argument, the imageName which is the image‘s name

This work is licensed under a Creative Commons Attribution 4.0 International License

inside AE.ImageManager. It also has an optional argument, which is the engine, which define on which AethelmE engine the image will be taken from and the sprite will be saved to. For sprite manipulating, there are several attributes exist inside this class, such as the angle which define the sprite‘s rotation angle from the center. Unlike the mathematical angle, this engine‘s angle zero is pointed to Y- positive-axis, or number 12 on a clock. When it‘s incremented, it will rotate clockwise just as a clock would.

AE.Sprite also provides attributes which automates common animations. The animationMode attribute will contain an animation constant which defines the animation type, while the currentAnimationMode attribute defines the current active animation. There are also animationFrames which is an instance of AE.Vec2 and defines from which frame to which frame the animation will play. For collision detection, AE.Sprite uses several methods subsequently. It support distance -based collision and object bounding box (or Separating Axis Theorem) collision. These functions are saved inside the static array attribute collisionFunction.

To check the collision detection, AE.Sprite has the isHitWith() function, which takes another instance of AE.Sprite for the argument. This function does the collision detection by calling saved collision functions inside the static collisionFunction iteratively. First, it will call the distance-based collision function. When the result is false, the function returns the value false, since there‘s no need to continue the detection (objects that are far apart logically wouldn‘t collide at any sprite‘s part). If the returned value is true, then the detection continues to the next algorithm, which is the SAT.

Then, the final returned value would be whatever value returned from this last function.

AE.Sprite also has the destroy() function. It will remove the sprite from the stage visually and programmatically. It means that not only the sprite will not be rendered, but it also will be removed from the array inside the active AE.DisplayList object.

P. *AE.AnimationMode Class*

AE.AnimationMode is a utility class that contains all constants for AE.Sprite‘s animation attributes, especially animationMode and currentAnimationMode. When the value is set to animationMode, it represent the sprite‘s whole animation. It is different when the value is set to currentAnimationMode, because it only represents the current flow of animation. The constant NONE means that there‘s no active animation, while the value ONCE means the animation only run from the start to end and the animation stops. The constant LOOP makes the animation loops infinitely when the frame reached the end. The constant REVERSE means the animation is running backward (from end to start). Finally, the constant PING_PONG makes the animation loops forward and backward continuously.

V. Testing And Implementation

This research created two tests. The first is the engine‘s unit testing. It means that the test will make sure the classes‘ functionalities are working properly. The second test is to compare a similar game between multiple free HTML5 game engines and AethelmE. The aspect compared are the CPU and GPU usage. The target of this second testing is to compare the games performance on engines with single drawing viewport and engines which support multiple drawing viewports.

A. *Unit Testing*

AethelmE engine works on most major internet browsers. Tested using Google Chrome, Mozilla Firefox, Opera, and Internet Explorer 9, AethelmE supports error handling, external asset loading (using AE.Loader and AE.MultiLoader class), a vector wrapper class, image and text rendering on HTML5 <canvas> element, and also detecting both mouse and keyboard input using AE.Input class. The aforementioned features are fully compatible with all of the tested browsers.

The problem lies on the AE.Sound class, as the class is not properly working on several browsers. There are two main problems of this class. First, is because this class cannot detect audio format compatibility automatically. This problem can be seen on Firefox‘s sound unit testing. As seen on the Figure 8, Mozilla Firefox can properly play audio with supported format (signified from the green SUCCESS result), but the test for detecting incompatible format failed, or did not work to be precise. Normally, when the test failed, the result box‘s color should change to red and the word FAILED showed, but this is not the case with the above test. The default text inside the box was still shown, and there was no change at all, even when it was left after sometime.

When the test is conducted more in-depth using Firefox‘s debugging plug-in, Firebug, it turned out that when incompatible audio format is detected internally, the browser is automatically set for not downloading the file. This explains why the test freeze in the middle, because the end callback function is never called and the result() function was never executed, making the test hanged without any precise result. Since this problem is caused by the browser implementation, there‘s not much that can be done for AethelmE, except to modify the game engine to match how the browser behave.

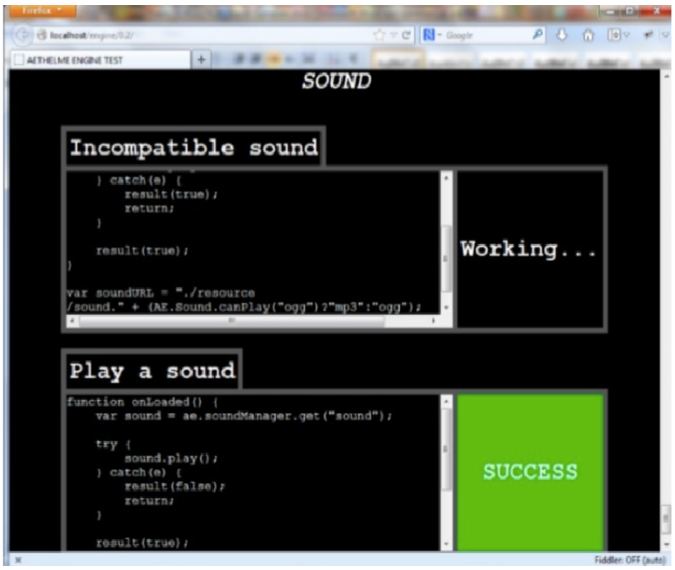

Fig. 8 Sound Unit Testing on Mozilla Firefox

The second problem of AE.Sound class is that it does not work at all on both Opera and Internet Explorer (e.g. no audio is played at all). As seen on Figure 9 and Figure 10, none of the audio unit testing are working on Opera and Internet Explorer 9, as there are no SUCCESS or FAILED message inside the result box. This no-result occurrence can only mean that (similar to the previous problem) the end callback is not called at all, making the result() function is not executed.

This work is licensed under a Creative Commons Attribution 4.0 International License

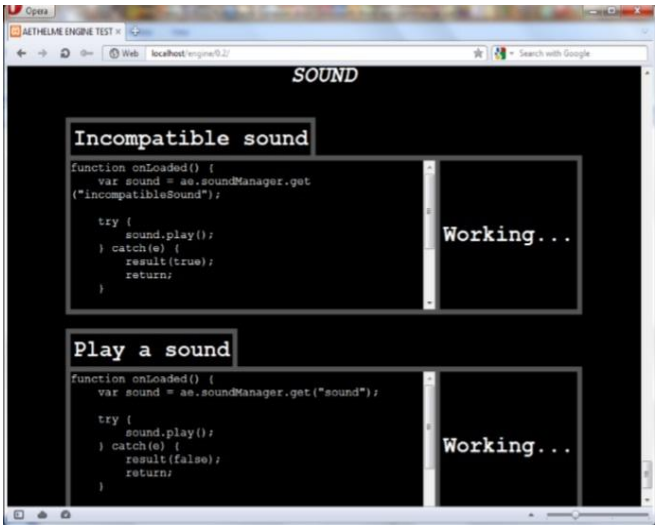

Fig. 9 Sound Unit Testing on Opera

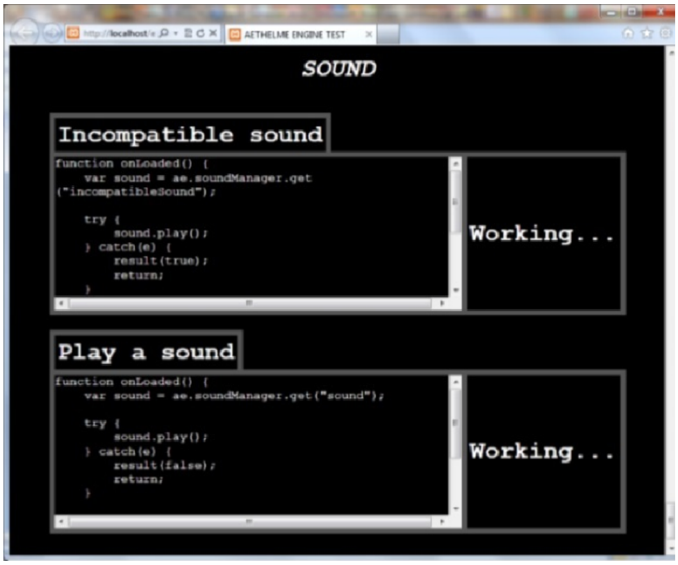

Fig. 10 Sound Unit Testing on Internet Explorer

B. Performance Comparison Testing

For testing the performance of multiple HTML5 game engines, this research focused more on comparing the performance on refreshing a single drawing viewport and refreshing two or more drawing viewports. The test was using ASUS Notebook UL80VT Series with the following specifications: 64-bit Windows 7 Home Premium, Intel® Core™ 2 Duo Processor SU7300 @1.3 GHz, NVIDIA GeForce G210M and 4.00 GB RAM

The game specification for this test is described as follows:

- A sprite which have four-frames looping walking animation. For engines which support multiple drawing viewports, this sprite will be put on the foremost viewport.
- A two-part background image which scrolls only to the right. For engines which support multiple drawing viewports, these background sprites will be put on the rearmost viewport.
- It should be able to retrieve user inputs.
- When right arrow is pressed, the walking sprite moves to the right. When it is already at the right edge of the viewport, scroll the background to the right.
- When the left arrow is presses, the walking sprite moves to the left. When it is already at the left edge of the viewport, do nothing.
- Record the CPU and GPU performance while the game is running in several states: idle, moving (walk), and scrolling background.

While the game seems pretty simple, this test can check how significant multiple drawing elements can impact the processor and graphic card‘s performance, because there are elements that should be updated every frame and there are some that do not need to be updated that frequently. This performance measuring is also dependant to browser‘s implementation of image rendering, both in DOM and <canvas> element, as different browsers can have their own way of rendering bitmaps on the screen.

Before started the testing (when the system on standby): 10% CPU usage and 0% GPU usage. Table 1 shows the result of the performance testing with three different states.

TABLE 1. Performance Testing Result

		Idle	Walk	Scroll
lycheeJS	CPU	33%	35%	35%
	GPU	28%	31%	28%
gameQuery	CPU	15%	25%	52%
	GPU	11%	25%	30%
Traffic Cone	CPU	45%	48%	58%
	GPU	33%	34%	30%
Crafty	CPU	34%	37%	77%
	GPU	45%	47%	36%
Construct 2	CPU	30%	35%	36%
	GPU	50%	35%	35%
AethelmE	CPU	31%	36%	34%
	GPU	29%	30%	32%

The Figure 11 shows the comparison of the CPU usage between the tested HTML5 game engines. Higher bar means more processor usage is observed for the specific engine, which is not good. AethelmE is able to have stabilized CPU performance when tested in multiple states, on par with LycheeJS and Construct 2. gameQuery has the highest performance when the player is idle and walking (means that the background is still static), but become much worse when the background is scrolling.

This work is licensed under a Creative Commons Attribution 4.0 International License

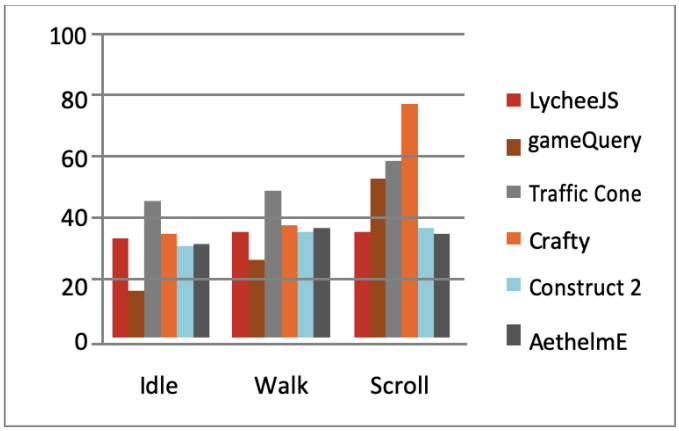

Fig. 11 CPU Usage Chart

Traffic Cone has the worst CPU performance in average, even though the engine also uses multiple <canvas> elements. Crafty has the same problem with gameQuery, whose performance worsens when the background is scrolling, and it is also the one that uses up the most CPU usage among the engines. Based on this test, it seems that multiple drawing elements are not the main aspect for having a performance gain compared to single drawing element. It might still help, but the result is not that significant.

Figure 12 shows the comparison of the GPU usage between the tested HTML5 game engines. Higher bar means more graphic card usage is observed for the specific engine. Please note that this result cannot be fully used to compare how good the engines are, because the GPU usage are dependant to the browser‘s rendering implementation. This graph only acts as a complement result for CPU usage comparison.

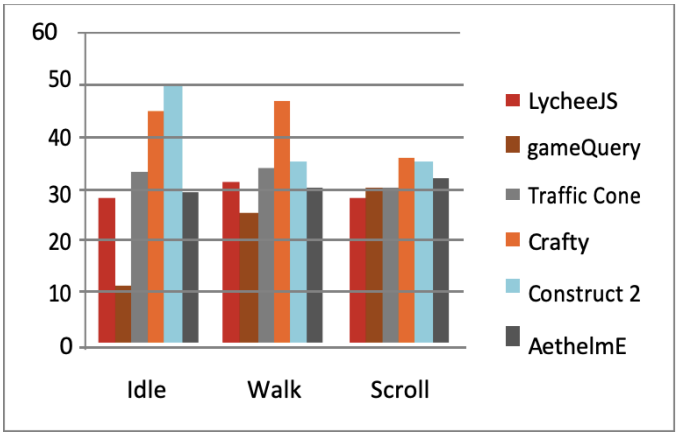

Fig. 12 GPU Usage Chart

As seen from the figure, almost all engines use generally the same amount of GPU usage, except for several cases. Crafty uses the most GPU usage in average, while gameQuery uses very little GPU calculation for idle state. Construct 2 has the most irregular GPU usage for the idle state. When the benchmarking game is started, Construct 2 uses only about 36% GPU usage, which is almost the same with other engines‘ usage. When the game is left for a while, the usage suddenly increased up to 50% and became steady on this number. This is not the case with the other two states, as it only has around 35% GPU usage.

VI. Conclusion

This research has successfully created a functional HTML5 game engine prototype which support for multiple canvas elements, called AethelmE. This research also did a performance comparison between several free HTML5 game engines and AethelmE, using a benchmarking game. Judging from the performance comparison result, it can be concluded that multiple drawing viewports (e.g. multiple canvas elements) does not have much significance in performance gain, because engines with single drawing element can have the same performance rate with engines with multiple ones.

References

[1] T.N.Sharma, P.Bhardwaj and M.Bhardwaj. (2012, Sept.). Differences between HTML and HTML5. International Journal of Computational Engineering Research. [Online]. 2(5), pp. 1430-1437. Available: http://www.ijceronline.com/papers/Vol2_issue5/AR02514301437.pdf

[2] B.Smus. (2011, Aug.). Improving HTML5 canvas performance. HTML5 Rocks. [Online]. Available: http://www.html5rocks.com/en/tutorials/canvas/performance/

[3] J.Park, "Study of network game engine technology for distribute processing," M.S. thesis, Dankook University, Yongin, South Korea, 2008.

[4] Unity. “Unity – Fast Facts”. http://unity3d.com/company/public-relations/

[5] M.Teschner, S.Kimmerle, B.Heidelberger, G.Zachmann, L.Raghupathi, A.Fuhrmann, M.P.P.Cani, F.Faure, N.Magnenat-Thalmann, W.Strasser and P.Volino. 2005. Collision detection for deformable objects. Computer Graphics Forum [Online]. 24(1), pp. 61-81. Available: https://hal.inria.fr/inria-00539916/document

[6] G.van den Bergen, “Efficient collision detection of complex deformable models using AABB trees,” Journal of Graphics Tools, vol. 2, pp. 1-13, 1997.

[7] S.Gottschalk, M.C.Lin and D.Manocha, “OBB-tree: a hierarchical structure for rapid interference detection,” Proceedings of SIGGRAPH, vol. 96, pp. 171-180, 1996

[8] D.Geary, “Essentials,” in Core HTML5 Canvas – Graphics, Animation, and Game Development, Crawfordsville, IN: Prentice Hall, 2012, pp 1-7.

[9] S.Fulton and J.Fullon, “Working with Audio, in HTML5 Canvas”, 2nd ed. Sebastopol, CA: O’Reilly, 2013, pp 381-387.

[10] E.Castro and B.Hyslop, “Introduction in HTML5 and CSS3”, 7th ed., 2012, Berkeley, CA, Peachpit Press, 2012, pp xvi.

[11] Crafty, “Crafty - Creating your first Crafty game”, 2010, [Online]. Available: http://craftyjs.com/tutorial/bananabomber/create-a-game#components

[12] C. Martens, “lycheeJS - Getting Started: Setup lycheeJS-ADK”, [Online]. Available: http://martens.ms/lycheeJS/docs/guide-setup-lycheeJS-ADK.html

[13] J. Mordetsky, “Trafficcone”, 2012, Available: http://github.com/j03m/trafficcone

[14] IBM Developer Works, Available: https://www.ibm.com/developerworks/ library wa-canvashtml5layering/

This work is licensed under a Creative Commons Attribution 4.0 International License