Masarykova univerzita
Fakulta informatiky

SVG based engine for rendering isometric graphic

Master's thesis

Bc. Vít Svoboda

Brno, spring 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Vít Svoboda

Advisor: Mgr. Marek Grác, Ph.D.

Acknowledgement

I would like to thank my advisor for patience and academic guidance. I would also like to thank Ing. Ondřej Žižka for technical consultations.

Abstract

This thesis examines the possibilities of using the HTML5 `svg` element for drawing game graphics. It describes an implementation attempt that would result in a graphics engine able to draw an isometric tile-based game in the `svg` element. Along with that it also summarizes differences to the HTML5 `canvas` element method of use. Finally, using the proposed implementation, assesses the usability of the `svg` element in games. Considering the performance differences across various web browsers, the `canvas` element, usually used for that purpose, seems to be a better option.

Keywords

2D graphics, game engine, HTML5, isometric projection, JavaScript, sprites, SVG

Contents

1 Introduction

Specification of HTML5 introduced two new elements to displaying dynamic graphics natively in the browser: `canvas` and `svg`[1]. Before that, browser games had to be built using third party solutions to display the game graphics[2]. With the broad adoption of the new HTML standard, game development focus shifted to the `canvas` element[2]. The question this thesis attempts to answer is whether it is possible to utilize the latter of the new elements in the same way as the first one has successfully been so far. Chapter 2 briefly describes the current state of graphics engines usage in games and expectations on provided functionality. Furthermore, the chapter focuses on a graphics style named after the projection method used – the isometric graphics and outlines its origin and usage in the course of video game history. This method can be applied in the web environment to achieve balance between visual appeal and rendering speed. At the end the chapter elaborates on the specifics of web graphics and the way they are usually presented to the users' clients.

Chapter 3 describes a graphics engine that would allow rendering of isometric game graphics to the `svg` element and could prove such usage is possible. This method of rendering graphics could serve as an alternative to the commonly used `canvas` as well as making certain aspects of the game development substantially easier. Chapter 4 summarizes technologies considered for use within the engine implementation. The actual implementation and its consequences for the engine user – a game developer – are described in Chapter 5. Finally the proposed engine implementation is subjected to testing and stress to assess the usability by real applications as described in Chapter 6.

2 State of the art

Modern video games are designed to be data driven[1]. That way as much code as possible can be used again in similar games. The common functionality can be packed in a software component called the game engine[4] and distributed separately from the game itself to other developers. The game engine is responsible for collision detection, physics calculation, game logic script execution and, for the scope of this thesis, most importantly, rendering. A part of the game engine that encapsulates the last mentioned responsibility is called graphics or rendering engine.

2.1 Graphics engine

Graphics engine is a software component responsible for transformation of a logical model of a scene to an image on the screen. Graphics engines used in native applications heavily depend on middleware like OpenGL or DirectX[4].

2.1.1 Typical functionality

The input for a graphics engine is a set of objects on the scene, data representing their appearance[2] and a camera position. The engine has to determine what objects or their parts are currently visible from the point of view of the given camera. A properly placed visual representation needs to be created for all of these objects. The output is a bitmap stored in memory, containing graphic representation of all the related objects. The bitmap can be prepared directly, or more commonly, created via a series of calls to some underlying API[3], rather than by manipulating the memory directly. Optionally, the engine can perform various post-processing on the output, such as anti-aliasing, shading etc.[4] However a more complex post-processing is usually performed only in 3D applications. When rendering a real-time image, the process has to be repeated several times per second to maintain a perception of a fluid animation without stutter. Simply put, the higher the frame-rate[4], the better[5]. Nowadays the frame-rate is limited by standard

1. Data-driven program determines execution flow based on provided data. Simply put, the data describes program behavior. Thus an unchanged code can perform different tasks when given appropriate data.[3]
2. Sprites, meshes and textures
3. Application Programming Interface
4. Frame rate is often referred to as FPS (Frames Per Second) with value equivalent to frequency in Hz.

display hardware with 60Hz refresh rate to 60 FPS, but new hardware capable of displaying images at 120, 144 or more Hz has been emerging on the market in the last few years.

2.2 Isometric graphics

There are many ways to project a 3D scene to a 2D screen. One of the older and less accurate is the isometric projection – an axonometric projection[5] with perpendicular direction of the projection and equal angle between the view plane and all axes[7] as shown in Figure 2.1. This way all 3 dimensions can be displayed sufficiently enough to allow orientation while keeping the rendering simple and therefore fast. However, perspective, an important property of real 3D projection is lost in the process and the distance from the camera has no effect on object size.

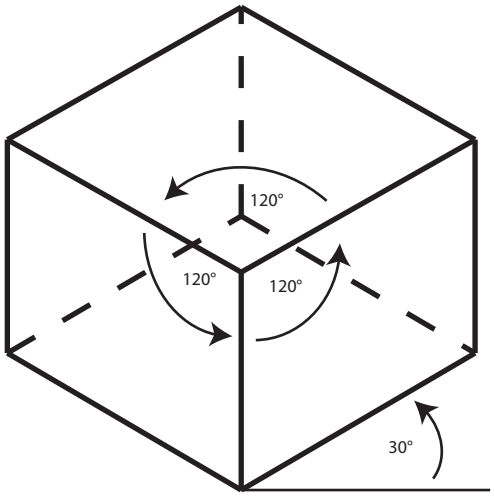

Figure 2.1: Isometric projection

2.2.1 Isometric camera

A characteristic attribute of isometric graphics is the camera orientation. The camera is top down, but tilted by 45 degrees in both directions. The

5. Projection of an object rotated along one or more axes relative to the plane of projection[6].

diagonals therefore should be at 30 degrees. However due to the irregular aliasing of these diagonals on the bitmap grid, the practical approach is slightly different. Figure 2.2 shows the problematic bitmap grid as well as the common solution. Instead of calculating the projection in the tilted camera, the image is rotated by 45 degrees and then the aspect ratio is changed to 2:1[2]. The result is a regular 2pixel stairs forming diagonals in about 26.565 degrees[6]. The welcomed side effect of this change is the complete elimination of trigonometry from the projection mathematics, further improving the rendering performance.

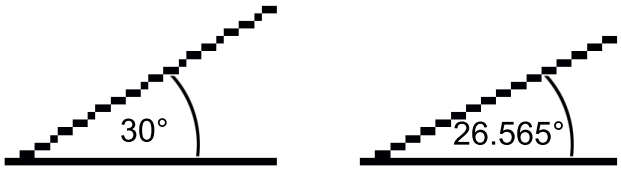

Figure 2.2: Isometric projection angle

2.2.2 Example games, brief history

The first game to use isometric graphics was *Zaxxon*, released by SEGA in 1982[8] (see Figure 2.3). This new approach allowed to display 3D scenes without the need for hardware accelerated 3D rendering. The same graphic style was later used in many other games, especially genres where the player needs to overview a large scene from the top. While *Dune II* released in 1992[9] still used direct top-down camera, *Warcraft: Orcs & Humans*, released two years later[10], tilted the camera slightly in one direction to improve the visual appeal. Later games in the strategy genre[7] already fully adapted the isometric projection. These strategies often divided the map to regular tiles that on one hand limited the player actions but also made collision detection and other game mechanics very simple. This design outlived the slow hardware with no graphic acceleration and is still popular both among independent game developers with limited resources and in environments where hardware acceleration is not granted[8]. When 3D acceleration

6. $\arctan \frac{1}{2} \approx 26.565°$
7. 1990s classics including turn-based strategy *Civilization II*[11], real-time strategy *Age of Empires*[12] and business simulator *Transport Tycoon Deluxe*[13]
8. Typically browser games discussed later.

became affordable, the focus of game development shifted to proper 3D rendering. Although the player could be given full control over the camera, many games keep locking the camera in the same angle as with the isometric camera in order to avoid player confusion. Provided combination of realistic look of 3D rendered scene with perspective, shadows etc. and a clear overview of the scene defined by the isometric camera angle has proven to be optimal for wide array of games. Prime example be professionally played *Starcraft 2*[14] – although it does allow camera rotation, the camera automatically returns to the original position soon afterwards.

Figure 2.3: Isometric graphics in original arcade version of *Zaxxon*[8]

2.3 Web graphics and HTML5

Web applications can only rely on the client's browser application and its capabilities. No middleware directly accessing the hardware is guaranteed. Therefore browser games[9] draw graphics to some element in the HTML page. Prior to the version 5 of the HTML standard, the games almost exclusively used Flash and required the client browser to have Adobe Flash Player plug-in installed[15]. With the broad adoption of HTML5, new options presented themselves, facilitating elimination of the dependency on 3rd party plug-ins. Among others, the two new elements to display dynamic graphics – `canvas` and `svg` – allow applications to stop making any assumptions about the

9. Games played on a web page displayed in the browser.

client other than regarding the support of HTML5[1].

2.3.1 Canvas, usage in games

The `canvas` element is very similar to the canvas objects in other programming environments[10]. It directly allows drawing to the canvas bitmap using JavaScript functions. Even though this approach is straight–forward, some complications arise when it is applied. The most noticeable being that when an object is being changed, everything covering it merely in part has to be re-drawn as well even if it stays unchanged on its own. Furthermore when drawing several objects that appear behind and in front of each other, these need to be carefully drawn in a correct order so the proper overlaps are maintained and the illusion of space remains unbroken.

Due to the similarity to other environments there is already large variety of game engines implemented based on the `canvas` element. Used underlying element is often reflected in the engine name[11], but other take `canvas` usage as a matter of course[12].

2.3.2 SVG

The `svg` element is very different as it displays the given SVG[13], a vector image described in a markup language of the same name. It can also be dynamically manipulated with JavaScript, but the approach is fundamentally different. Elements are added to the DOM[14] as if only standard HTML was used. Instead of redrawing the whole image or a part of the screen, individual objects can be added, removed or changed. This makes operations like object animation much more straight–forward. On top of that, user interaction handling is easier as well since the event handlers can be hooked onto the elements representing the actual objects rather than calculating what was placed on the mouse click location.

Even though browsers used to support static SVG for a long time, not until HTML5 did they support dynamic in-line SVG[1]. This is now available with the previously mentioned element. However, since rendering the markup language is much more complicated than simply dumping the canvas bitmap on the screen, the browser support is delayed and also more sensitive to

10. E.g. classes `System.Windows.Controls.Canvas` in .NET framework[16] or `java.awt.Canvas` in the Java platform[17]
11. E.g. *Canvace*[18] or *Canvas Engine*[19]
12. E.g. *enchant.js*[20] or proprietary *Isogenic Game Engine*[21]
13. Scalable Vector Graphics
14. Document Object Model

performance problems. Drawing to the canvas may take a lot of JavaScript execution time, but in the end the display of the element takes amount of time only proportional to the size of the canvas, not complexity of its content. Creating a large SVG will take similar time to drawing to the canvas, but then rendering it on the screen will take the browser more time depending on the number of elements rendered.

3 Suggested solution and its design

As was discussed earlier, `canvas`-based engines for browser games are quite common. Why is that there are no game engines based on `svg`? W3Schools in a chapter about HTML5 `svg` element warn that, unlike the `canvas` element, `svg` is not suitable for game applications[22]. However is it possible to use an inline SVG to draw game graphics instead of the standard canvas approach? As proven by a demo build using Raphaël[1], at least a small game with simple graphics actually can be done[23]. With the event handling support, SVG should be even more suitable for interactive applications and additionally the possible manipulation only with specific objects in a drawn scene should make the subsequent scene updates much easier.

The proposed idea based on everything mentioned so far is to make an attempt to create a graphics engine using the `svg` engine as a rendering platform. An engine that will be able to request data that describe the area currently on the screen, transform the retrieved data to their visual representation and render as SVG elements. Of course, all of this and the user interaction handling needs to be separated from the actual implementation for the engine to be generally usable by different end applications. To keep the scope focused enough so the implementation can provide some actually helpful functionality for the end application developer rather than just another abstraction layer over the SVG DOM, the engine should be limited to render isometric tile-based graphics that are, as discussed earlier, still popular among both developers and players. This limitation allows the implementation to encapsulate all logic regarding isometric coordinate transformation as well as the tile update handling.

3.1 Data retrieval

Although in case of a single-player game[2] the data can be completely generated and stored on the client instance, remote data retrieval needs to be supported as well. Especially since even single-player games are likely to store level map data in some common storage, the remote data access is expected to be prevalent use in eventual implementation usages. To keep the data transfer down, the data provider[3] should be given an option to decide either to send full data matrix of requested area in the response or send only

1. An SVG manipulation library discussed later.
2. Game for only one player with no interaction with other people playing the same game.
3. Hereinafter referred to as the server.

data of the tiles that changed since last request in case the matrix of actually useful data would end up being too sparse. The former approach needs less data per tile since only the coordinates of a top left corner of the requested area needs to be included in the response since all other coordinates can be calculated from the rest of the response data. On the other hand, the latter approach needs coordinates for each tile. Of course the efficiency of both ways depends on many other things. How thick or thin the client is and therefore whether the server keeps track of individual client request history. The tile data size to the two-integer coordinates ratio is also important as is the volatility of the map data. Nevertheless, both approaches find its use and the engine has to be able to process data presented in both formats.

The tile data record may contain more information than it is necessary to deduce the visual representation of the tile. Hence only a subset of this data record should be passed to the general map data requests to further lower the bandwidth usage. Then there must be a way to retrieve the rest of the tile data record for game logic purposes though. To provide this channel, engine has to implement a full data request for a tile involved in advanced operations.

3.2 Sprites

The standard way of making 2D games is to draw small images of objects and then compose the scene using these images. This way the object can be animated easily by replacing the image, visually representing the object, with next animation frame image. For performance reasons all these sprites are compiled to a single image called sprite-sheet. This way all sprites can be loaded in a single operation as opposed to reading multiple files from the file system or, even worse, multiple HTTP requests over the network.[2] To support this optimization, the graphics engine needs to be able to draw a sub-selection of the sprite-sheet image. When working with HTML5 `canvas` element, JavaScript function `CanvasRenderingContext2D.drawImage(img, sx, sy, swidth, sheight, x, y, width, height)` does exactly that[24]. First set of coordinates and dimensions determines which part of the source image will be drawn. Second set of coordinates and dimensions then determines the target location on the canvas.

Unfortunately, when using SVG, no such a straightforward and convenient way is available. To display a section of an image, one needs to create a pattern from the image and position it using a negative coordinates. The pattern must be configured to use *objectBoundingBox* units so it doesn't shrink the whole sprite-sheet in a single target element. Then an SVG el-

ement (usually polygon) using this pattern can display desired selection of an image in its background. The limitation of this approach is that all elements using the same pattern will have synchronous animation if the pattern gets animated. That usually doesn't look too organic in the result, on the other hand it nicely emulates the early 90s era of gaming when asynchronous animations were too performance expensive.

The more natural approach for `svg` would be having sprite-sheet defined as a separate SVG file containing vector sprites defined by SVG markup. Although this is certainly possible, vector graphics are not as widely used as bitmaps in the game development since creation of detailed vector assets is more complicated compared to the bitmaps.

Unlike `canvas`, the SVG also allows to use animated GIF[4] images as sprites. The creation of sprites or even sprite-sheets this way is more complicated. However when using sprites that are already animated, execution time needed to animate a collection of static sprites can be spared. The amount of saved time of course depends on the browser implementation and its support of GIF as an image format. In some cases this may even yield worse result than using static image formats.

4. Graphics Interchange Format is an image format that allows to contain a series of images that are displayed in sequence to show an animation[25].

4 Used technologies

An implementation of the described engine would integrate multiple libraries and software components as well as technologies implemented by the browsers as part of various standards to achieve the best results without attempting to reinvent the wheel. All of the technologies are selected to address a specific need. The following enumeration outlines the rationale behind selecting those worth mentioning.

4.1 Engine itself

The graphics engine will be run on the client and therefore all technologies used in it are based around JavaScript. Either as a JavaScript API provided by the browsers themselves or various JavaScript libraries offering functionality useful in some aspect of the engine implementation.

4.1.1 Browser support of HTML5 SVG element

HTML5's `svg` element as such is well supported by all modern browsers[22], but there are some difficulties with the implementation details in different browsers. A striking example being that Internet Explorer entirely lacks support of `foreignObject` SVG element that normally allows the nesting of HTML in SVG images[26]. While the completeness of the standard implementation differs across the board, it tends to slowly improve over the browser release iterations and use of the element is fairly safe across all currently available browsers.

4.1.2 SVG manipulation JavaScript libraries

The contents of the `svg` element can be manipulated directly via JavaScript functions implemented on the element and its possible children by browsers as a part of the standard[27], but like regular HTML DOM manipulation, it is laborious and often ends up with lots of repetitive code. To eliminate this drawback, there are several JavaScript libraries that provide abstraction layer over the SVG markup. All of the common ones are open source.

A pioneer among these libraries is *Raphaël*. The projects puts emphasis on wide browser support including archaic Internet Explorer version 6.0 [28].

Raphaël's complete rewrite, *Snap.svg* is not held back by sustaining compatibility to obsolete browsers and so it can support features like masking, clipping, patterns, full gradients and groups[29]. *Snap.svg* is quite popular

among the front-end developers who typically create animations based on a static SVG for web sites[30]. When working with SVG elements, their attributes are represented as string values. This may be perfect for the mentioned application, but when creating all of the content dynamically, it involves a lot of string concatenation in the end, which is far from ideal in the intended use.

Next library, simply called *svg.js*, takes more a object oriented approach[31] that is more suitable for the graphics engine. The project is modularized so a production environment can be built with only the functionality that is really used to decrease script size, thus the amount of data required to be downloaded to the client prior to the application start. There are also quite a few additional modules providing extra functionality that might prove useful.

Another popular project called *D3.js* does not limit its scope to SVG and approaches the DOM manipulation in a similar manner to *jQuery*[32]. Emphasis on querying the DOM, that is very performance demanding once the DOM has a large amount of elements, makes this project not suitable as well as some of the previously mentioned.

Of course there are other libraries providing similar functionality, but since the game developer using the engine will have to work with given SVG manipulation library to some extent, commonly used library with good documentation may make his use of the engine much easier. All considered, the *svg.js* project was selected to serve as a SVG markup manipulation API.

4.1.3 General utility JavaScript libraries

Other general purpose JavaScript libraries are used to keep the actual engine code clean and readable. Like the previously discussed libraries, these are open source as well and use non-restrictive licenses.

To help manage the JavaScript application structure, a library allowing to define JavaScript objects as modules and subsequently the dependencies between them that is called *require.js* is used. During execution on the client it makes sure all files containing the required modules are loaded prior to their usage. [33] This approach eliminates many flaws of JavaScript as a language, particularly the declaration scope confusion that leads to possible redeclaration of variables and dependency on functions or variables that may not be declared yet.

Some very limited HTML manipulation is simplified using the *jQuery* library. Since the engine does not work with HTML outside of the `svg` element, the usage is rather minimal, mostly accessing element properties in a

unified fashion[1]. Since the display of game scene can generate quite a lot of elements, querying them, the main point of *jQuery*[34], has to be done with performance in mind. On the other hand should the developer choose to use the `foreignObject` element to create the game UI as nested HTML, *jQuery* can make it very easy.

Automatic tests of the engine API are written using the *Jasmine* framework. It allows to define test cases and expected behaviour[35]. Out of the box, these tests can be run in the browser just like the engine itself. Furthermore, when using the *Karma* test runner the tests can also be integrated to the *maven*[2] build process[37].

4.2 Other technologies used in the demo game

The demo application created to demonstrate the capabilities of the engine combines it with other technologies just like a real world application would. Other application may use completely different set of dependencies though, therefore the ones used in the demo application are in no way bound to the actual engine.

4.2.1 JavaScript objects provided by browsers

Since the engine itself is abstracted from actual data source and format of the retrieved data, the browser support for network communication and data serialization is not needed until the specific game implementation.

To obtain data from the server the demo application uses `XmlHttpRequest`, a JavaScript object widely supported by browsers[38]. The object allows the creation of an asynchronous HTTP request with a callback when the response is obtained.

To parse the JSON[3] responses to JavaScript objects the `JSON` JavaScript object specified by ECMAScript 5.1[4] is used. This object is also supported by modern browsers[40].

---

1. Even something as simple as getting element's actual width can be quite laborious to get working across browsers without *jQuery*.
2. Maven is a build automation tool[36] used in this project mostly due to the demo back-end written in Java.
3. JavaScript Object Notation data format
4. ECMAScript is a specification of a script language. One of its implementations is JavaScript[39].

4.2.2 RESTEasy

RESTEasy is a JBoss project allowing to create a web service with parametrized URLs mapped to custom methods[41]. This service can very easily provide RESTful[5] management API, hence the project name. Jackson JAX-RS providers[43] are used instead of the default JSON serializer to allow the service to return custom Java objects serialized to JSON without the need for any additional annotations. This framework was selected due to the simplicity of creating a service which would provide access to the pseudo randomly generated data model of the demo application.

4.2.3 AppEngine API

Google App Engine is a cloud hosting service provided by Google. The engine demo application was hosted on this service for demonstration purposes and so the project had to contain piece of appropriate configuration. With a generously limited bandwidth, the service is free of charge. [44]

5. RESTful service meets all requirements defined by Representational State Transfer architecture, that is mainly unified access to creation, reading, update and deletion of provided resource[42].

5 Implementation

An important factor of the engine implementation is reusability. This requires abstraction of all the details specific to any game. Anything that would lead to these specifics is delegated to objects providing the specific game implementation via calling their interfaces. Figure 5.1 shows dependencies among the engine JavaScript modules as well as the dependency to the external implementation of a specific game.

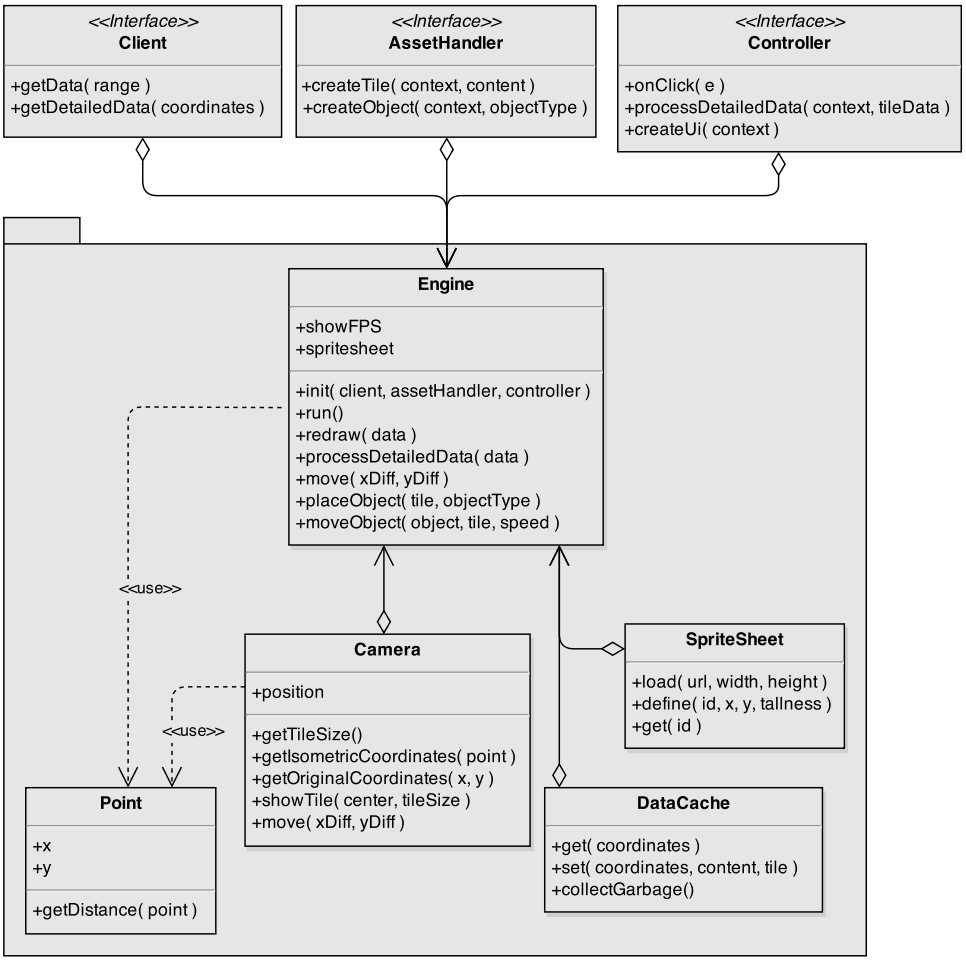

Figure 5.1: Diagram of the JavaScript modules

For the purposes of the engine, the game implementation is divided to three interfaces, responsible for different aspects of the implementation. The

Client interface is supposed to arrange all communication with the server while *AssetHandler* takes care of translating the data to their visual representation. The last interface, called *Controller*, is responsible for handling user interaction and game logic. Although the demo application implements each interface separately, nothing prevents the end application developer to implement all of the interfaces in a single object.

The sole purpose of the demo application is a demonstration of the engine implementation capabilities and apart from that it serves no practical use. With that in mind, the map is divided to regular tiles and each tile can be covered with different kind of terrain. The player can move around the map and place buildings and vehicles on the tiles, with very little constraint in sense of possible financial balance, tile terrain type or anything else. The vehicles can be moved to another tile by selecting the vehicle and then selecting the destination tile. On the other hand the buildings are stationary, but more objects can be placed on top of them. With further expansion of this very basic game logic, a game similar to the *Transport Tycoon Deluxe* or *Civilization* could be created. However, the additional effort required for that would add little value to the intended purpose – the engine demonstration.

5.1 Viewport population and data fetching

As discussed in Chapter 2.1.1, the main responsibility of the engine is to draw the game scene to the screen. In case of tile-based games, this involves data that describe contents of all tiles to be even partially displayed in the current viewport[1]. These data are periodically requested from the client object to keep the screen up to date. An asynchronous callback on the engine then handles the response data that can be a continuous array of tile content values or if the client determines that few enough tiles changed, only the content and coordinates of these changed tiles can be returned. This may save data traffic when repeatedly requesting updates of the same area as discussed in Chapter 3.1.

The client–engine interaction could have been implemented in a push manner, but then the client would have to take responsibility for the game update loop. Since the game update loop is clearly the responsibility of the game engine[4], engine is – in configurable intervals – polling the client for data updates.

Due to the isometric projection employed in the rendering, the range

1. Viewport is a viewing region, in this case defined by a `div` element passed to the engine constructor.

of tiles needed to populate the whole screen cannot be trivially defined. Figure 5.2 shows the minimum amount of isometric tiles needed to cover a rectangular viewport. The dark grey tiles are not fully in the viewport, but still need to be rendered. To keep the data request simple and small, the white tiles are requested as well. Therefore, the request may consist only of a top-left corner coordinates and then either the width and height of the requested area or the bottom-right coordinates. These options are mostly equivalent and the was chosen for the engine implementation.

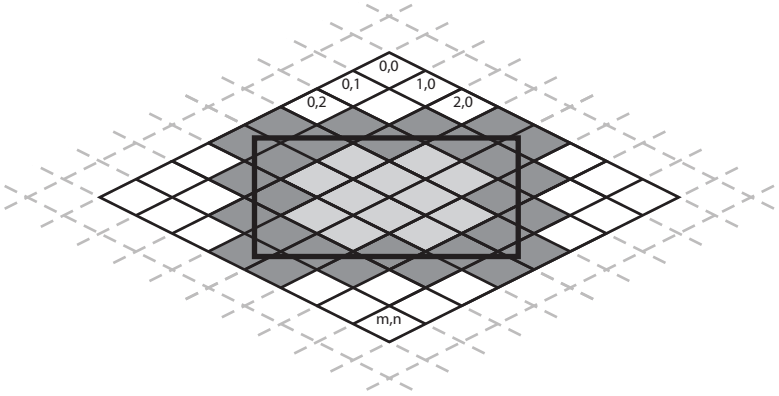

Figure 5.2: Data needed to fill a rectangular viewport

5.1.1 Data caching

When the player moves around the map, server is requested for data on respective locations, but the response is not likely to be immediate. To make the player experience smoother, the previously obtained data can be stored within the engine in case the player moves to the location of these data. Then, the potentially out-of-date tiles can be displayed while the new data are obtained. Excessive data obtained with each response, discussed earlier,can be also utilized using this approach.

Storing data on the client involves a couple intricacies. These are connected to the behaviour of the JavaScript array object and its low level implementation. The problematic part is that only non-negative integer indexes are supported by the array[39]. Everything else – be it negative integers, floating point numbers or straight up strings – falls back to the property indexer that is not only much slower than the array indexer, but also requires

specific iteration. The tile coordinates are integers, but from the $(0, 0)$ center spread to all four directions, making half of them negative. Fortunately, this can be eliminated by a simple transformation of the coordinates prior to the storage. Using the function declared in Figure 5.3 the negative indexes are interspersed with the positive ones and the zero in the same manner a Turing machine with two way infinite tape can be simulated on a Turing machine with a tape infinite only in one direction.

$$f(x) = \begin{cases} 2x & \text{if } x \geq 0 \\ -2x - 1 & \text{if } x < 0 \end{cases}$$

Figure 5.3: Index transformation function

Array in JavaScript is natively sparse, therefore missing items take up close to no memory[45]. However collecting all obtained data without a thought is out of question since as a wise man once said, "a cache with a bad policy is another name for a memory leak"[46]. All the obtained data can be cached as long as they are deleted at some point. The data cache implementation used in the engine uses two-dimensional arrays to store the obtained tile data as well as the respective SVG element as long as the tile is within the viewport. Once the first array is filled to a given limit, the second is filled instead and reading tries the current array first and if that fails, falls back to the older array. Once both arrays are filled to the given limit, the older one is deleted and serves as an empty current array while the previously current one is retired. This mechanism is not very sophisticated, but does the job well enough.

5.1.2 Isometric transformation

All functionality linked to the isometric projection is encapsulated in the *Camera* object implemented as a part of the engine. The transformation of orthogonal coordinates to the isometric ones defined in Figure 5.4 needs to be modified to also take into consideration the fact that the data coordinates are in virtual tile units while the isometric coordinates used on screen need to be in pixels[2]. Since the tile size in pixels is known, this can be achieved with a simple multiplication. Another necessary modification is to center the whole transformed image to the screen by subtracting a half screen size offset.

$$isoX(x, y) = \frac{x-y}{2}$$

$$isoY(x, y) = \frac{x+y}{2}$$

Figure 5.4: Isometric coordinates transformation

5.2 Object stacking and verticality

Just like when working with the full 3D projection, objects that are further away need to be hidden behind objects closer to the camera and objects that are on top need to hide objects beneath them. Otherwise the perception of space would be completely ruined. When working with a `canvas` element, this needs to be handled by properly ordering all objects to be rendered so the objects in the foreground are drawn last and cover anything drawn prior to them before the actual rendering starts. When also considering the elevation of each element, this may become quite a complex operation.

Fortunately a DOM manipulation – be it SVG or standard HTML – allows for a different approach. Standard DOM element function `insertBefore` allows to add element to a specified place in the element collection[47]. Since the elements are rendered in the order in what they appear in this collection of sibling elements, the last remaining problem is to find where in this collection the new element should be placed. As long as a position on the map is stored with each tile element and each object element keeps the information about the tile the object is placed on, the place where the new element will hide everything that is behind while being covered with everything that is actually in front of it can be found. The indicator used in the engine implementation is a sum of both x and y coordinates for it correlates with the y coordinate of isometric projection as is apparent in Figure 5.4. If all elements are added using this procedure, the collection is naturally ordered in the same manner as it would have to be ordered prior to rendering to the `canvas` element. It is also worth noticing that objects capable of position change need to be moved in the element collection during the movement. Otherwise an object that was correctly covering another object may end up still covering it even though it should not anymore. Figure 5.5 shows proper ordering change during a movement of sample objects, in this case a train car.

Objects elevated by either standing on something or flying in space need to be handled extra in the isometric projection. Their altitude needs to be reflected in the final coordinates by a y-coordinate decrease. The engine calculates the amount of this decrease as a sum of heights of all objects that

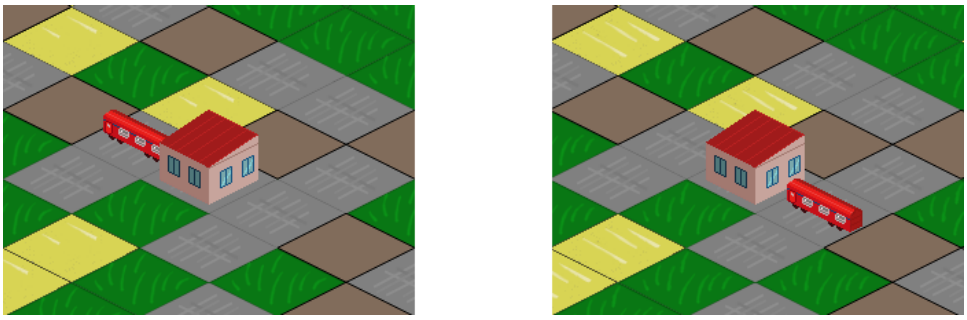

Figure 5.5: Object movement handled by the engine

are beneath the given object on the same tile. Stacking objects on top of each other is shown on Figure 5.6.

5.3 Asset declaration

Whenever the engine needs to create something with visual representation, interface referred to as the asset handler is consulted. The asset handler functions are passed the SVG context in a parameter and are expected to create an SVG element according to the object type or content passed in other parameter. The engine then makes sure the result is added to the appropriate place in the DOM and positioned to fit a location in the isometric projection.

This would cover the necessary engine functionality, but one more thing is implemented to make the end user's life easier. When creating the element to visualize some data, the visual representation needs to be obtained from somewhere. To manage these in a form of bitmap sprites, an object called *SpriteSheet* is made available. During the initialization of the game, one or more images can be loaded in memory and these or their parts can be declared as static or animated sprites. Each sprite is given an identifier so it can be later easily retrieved and used during object creation. The sprite declaration needs to know the position of the sprite in the potentially larger sprite-sheet and optionally also the number of animation frames and the speed of the animation, or to be more precise the delay after each animation frame. Figure 5.7 shows a sample sprite-sheet with three sprites, first is a desert with five animation frames, second is grass with only three animation frames and the last is static concrete with no animation. Usage of this sprite-sheet can be seen on the background of Figures 5.6 and 5.5. When the input image is an animated GIF, no animation needs to be declared with the sprites

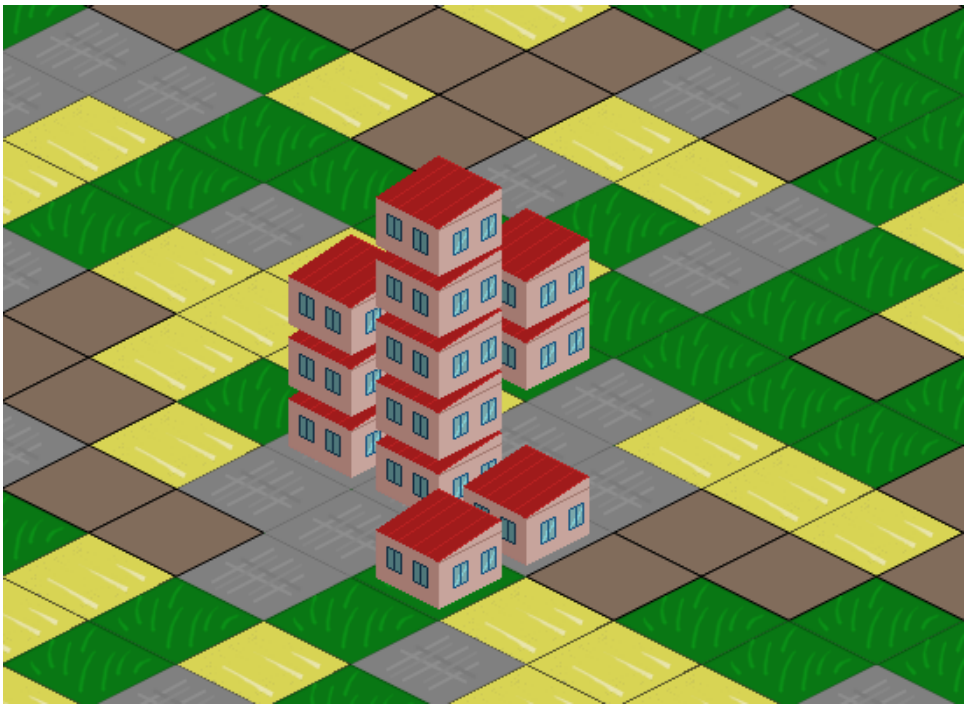

Figure 5.6: Object stacking handled by the engine

since it is performed by the browser, not by DOM manipulation performed by the engine.

The *SpriteSheet* is just an optional tool that can be ignored if the end application does not use bitmap sprites. In that case the SVG context is fully sufficient for the developer to create anything they may need.

5.4 Update loop

Even though the user interaction can be handled by asynchronous calls, a game engine still needs an "infinite" loop[2] to update the game state according to external powers, game rules and to handle animations. In JavaScript, the update loop can be implemented basically in two ways. The straightforward way would be to set an interval and handle the game update in the callback. However, using native interval is deprecated since the introduction of the `requestAnimationFrame` function[48], the latter and a better way. The scheduled interval cannot be synchronized with the browser up-

2. A loop running for the whole duration of the game.

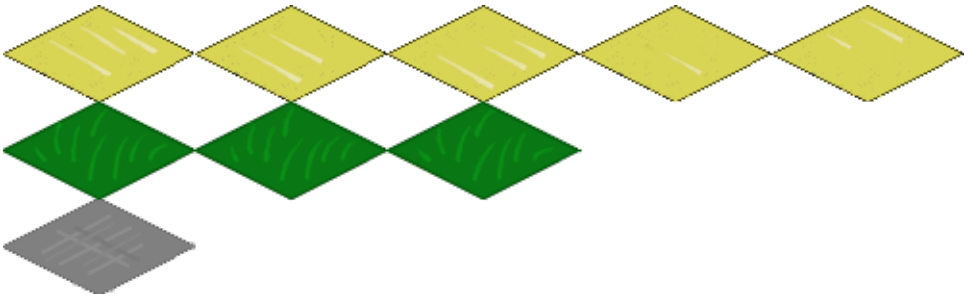

Figure 5.7: Sample sprite-sheet

date frequency and usually leads to unnecessary updates, especially when the browser window is not visible[49]. Using `requestAnimationFrame` the update callback obtains a timestamp of the moment when the update was executed and can be compared with the same information obtained by a previous update to calculate the refresh rate or evaluate the necessity of server polling or sprite animation frame change. Since the engine needs a continuous loop, every update method needs to queue another animation frame request[48]. All of this is encapsulated in the *Engine* object and everything the game needs to do after it is initialized is to call the `run` method.

5.5 User interaction

Handling the user interaction is delegated to the *Controller* object serving as one of the game implementation interfaces. The controller is responsible for user interface creation as well as handling mouse and keyboard events. The engine attaches the controller's `onClick` function to each tile created to make the user interaction handling easier. Keyboard handling can be implemented in the controller and does not require any direct support from the engine.

To obtain additional data, the controller can entirely bypass the engine and call client directly. The result of detailed data request discussed in Chapter 3.1 is passed to the controller anyway since the graphics engine does not need the additional information for any of its functionality.

To demonstrate the engine capabilities the demo application implements a couple of sample objects that the user can perform various actions with. The train car can be placed and moved. The building can be placed and then serve as a platform for more objects to be placed on. Both can be seen on Figure 5.5. Even though so few restrictions given to the actions make little sense for a real game, actual game logic in the demo controller would be counterproductive, only obscuring the engine usage.

6 Testing and measurement

Testing JavaScript has its specifics, mainly due to the strong dependency on the client web browser. Even though many standards were defined in an attempt to unify the browser behaviour, the actual implementations adopt the standards slowly and often deviate in seemingly minor details. Therefore, the testing usually involves as many browsers as is expected to be commonly used among the target user base. One can be careful not to use anything unsupported by some on the target browsers, but at the end of the day, running the application in all of the browsers may still uncover some compatibility issues. Automation of this process can save quite a lot of time.

6.1 Automatic testing

Part of the build process of the project containing the engine implementation are automatic tests. The expected behaviour of most of the implemented functionality is verified by running all of these tests by the end of each build to detect regression of bugs. The automatic tests can easily check API behaviour in extreme cases that may be very difficult to reach from the user interface. The *Karma* runner discussed in Chapter 4.1.3 used to do the execution of tests defined in JavaScript can be configured to run the complete test suite in many supported browser environments, however usually only single environment is enough to discover discrepancies created by an incautious refactoring.

6.2 Performance in various browsers

One of more likely reasons for the `svg` element to be inferior to the commonly used `canvas` element in regards to the game graphics could be insufficient performance when working with a scene rendered as a large amount of SVG elements. In extreme cases, this could make the `svg` element completely useless for more complex games. To test this concern, a performance evaluation of the final engine has been performed on various platforms available at the time.

6.2.1 Basic test scenario

The primary test scenario consisted of rendering an area of given size covered with pseudo random distribution of static and animated tiles. The area size used for majority of the cases was 1920 pixels by 1080 pixels, which is equal to

the industry standard resolution of FHD[1] displays. Roughly a thousand tiles were needed to fill the area. Twice a second, a server update was requested resulting in content update of fifty tiles. After initialization of the scene, one minute was given to stabilize the frame-rate. Then a thousand objects[2] were added one by one with one second delay after each addition. An average frame-rate over each of these second long intervals was recorded with the current number of added objects. All of this was scripted in a similar manner to the automatic tests. The scenario was designed this way to simulate the normal usage by a player of potential strategic game. By the end of the scenario execution, the screen would look like shown in Figure 6.1. One execution of the described scenario takes roughly 20 minutes regardless of the used hardware due to the scripted delays. The machines used to run the performance tests were running Microsoft Windows 7 and were also set up to run web server hosting for the tested application and provide data updates.

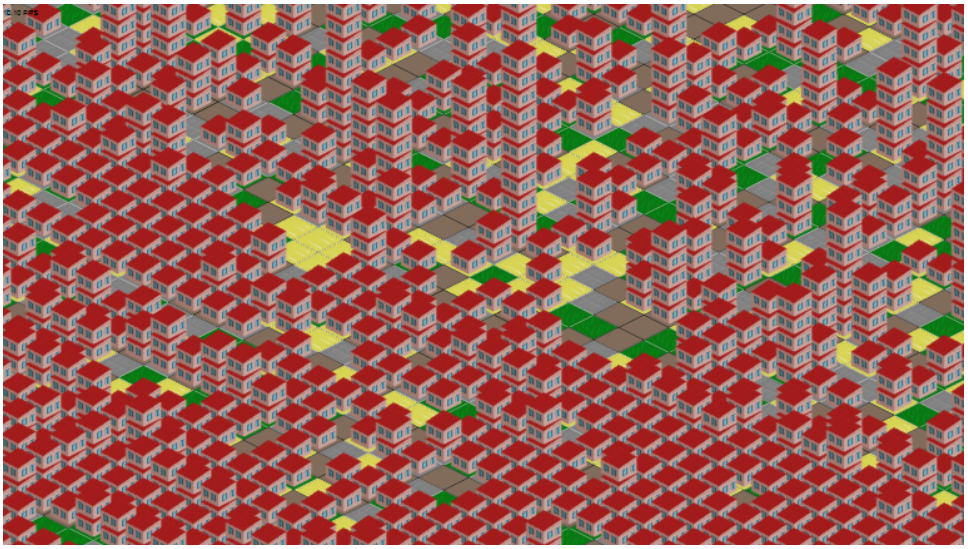

Figure 6.1: Screen depicting the end of the performance test scenario

The described scenario was run in various currently relevant web browsers, including the very common Google Chrome, Mozilla Firefox, Microsoft Internet Explorer and lately not that very frequent Opera. Out of curiosity a version of the Firefox browser compiled with optimizations for 64bit systems called Waterfox[50] and the recently announced Vivaldi available as a tech-

1. Full High-Definition
2. The building object visible in Figure 5.6 was used for this scenario, but any other object could have been used.

nical preview[51] were included in the browser selection as well. The ideal frame-rate is at 60 FPS or only slightly below that, but games are usually still playable down to 20 FPS[3]. Figure 6.2 shows the test scenario run results across all the tested browsers. The striking similarity of top three lines can be explained by Chrome, Opera and Vivaldi sharing the same layout engine – *Blink* that is a fork of *WebKIT*[52]. Substantial drop in frame-rate can be noticed when the screen has been filled with more than 300 objects. At that point the screen was already completely cluttered with objects. Both Firefox and Waterfox showed very unstable and borderline usable frame-rate. The Internet Explorer proved useless with this many elements on the screen.

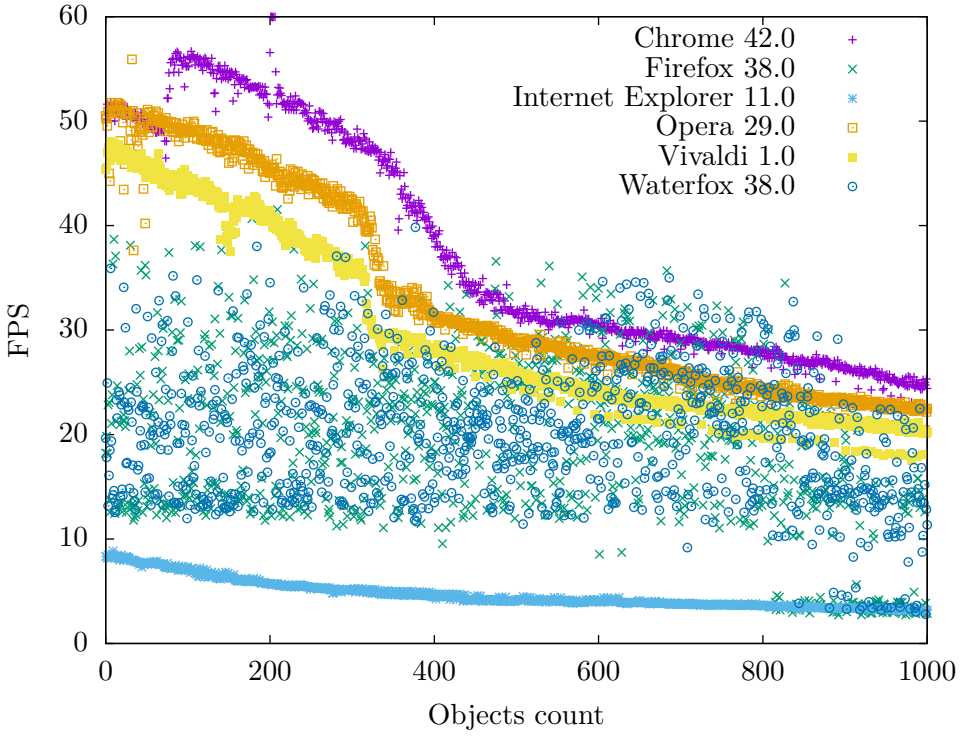

Figure 6.2: Performance test results across browsers

Worth noticing is also the browser hardware usage during the tests, all run on a 64bit desktop PC with 4core CPU Intel Core i5 2500K CPU at 3.3 GHz and 8 GB of RAM. None of the browsers deviated by excessive memory usage. However the WebKIT based browsers were spawning multiple

3. Depending on game genre – games that demand quick reactions require higher frame-rate to be played.

processes that all together used around 60 % of the CPU while Firefox steadily occupied only 30 % of CPU and the abstemious Internet Explorer consumed only around a quarter of the CPU time.

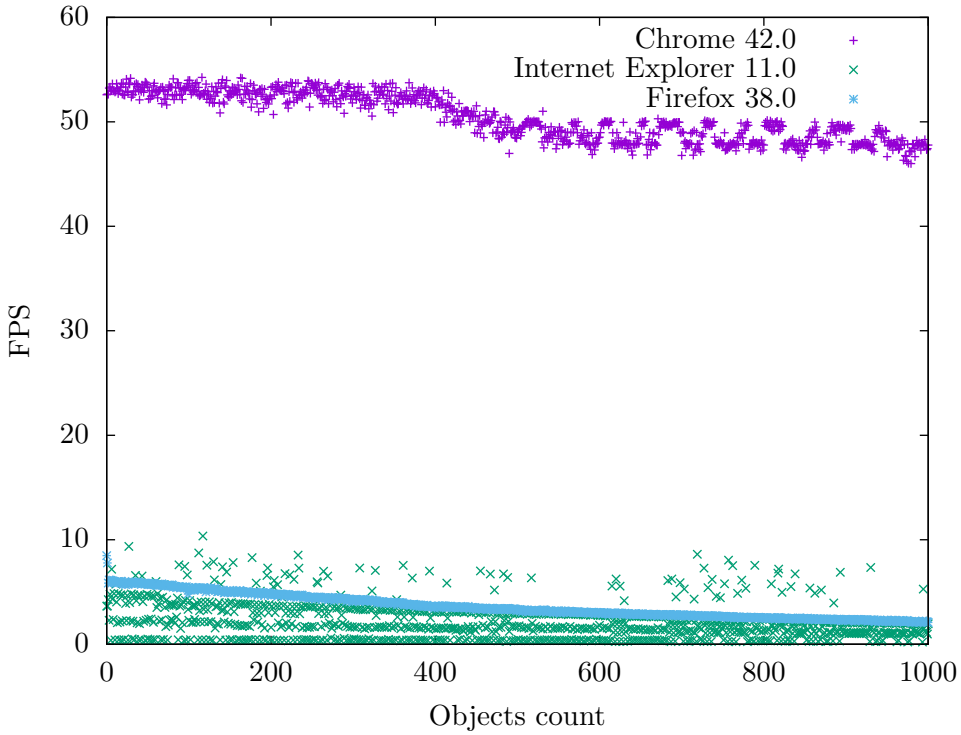

Figure 6.3: Results of the second test featuring animated GIF sprites

6.2.2 Performance impact of animated GIF sprites

Selected browsers were subjected to another test, using the same scenario but now using animated GIF sprites instead of handling the animation in the engine code. Since WebKIT based browsers proved to perform very similarly and there was no difference between Firefox and Waterfox either during the earlier tests, only Chrome, Firefox and Internet Explorer were subjected to the second test. Results are displayed in Figure 6.3. While the frame-rate drop after a certain amount of objects has been added to the scene present with Chrome during the previous test was eliminated, earlier all over the place Firefox frame-rate fell just above the Internet Explorer level. These changes in the performance suggest that using standard sprites and letting

the engine animate them in the JavaScript code would be a better option in most cases.

6.2.3 Other performance influences

To compare how the engine performance scales with other properties of the environment, two more tests were performed. The first tried the so far best performing browser on a PC with considerably older hardware. The second testing machine was a laptop with 4 GB of RAM and Intel Pentium Dual-Core T3400 CPU at 2.16 GHz, equipped with, as the name suggests, only 2 cores. Figure 6.4 compares results of the previous machine and the slower laptop. Using the older hardware, Chrome managed to keep frame-rate only barely usable.

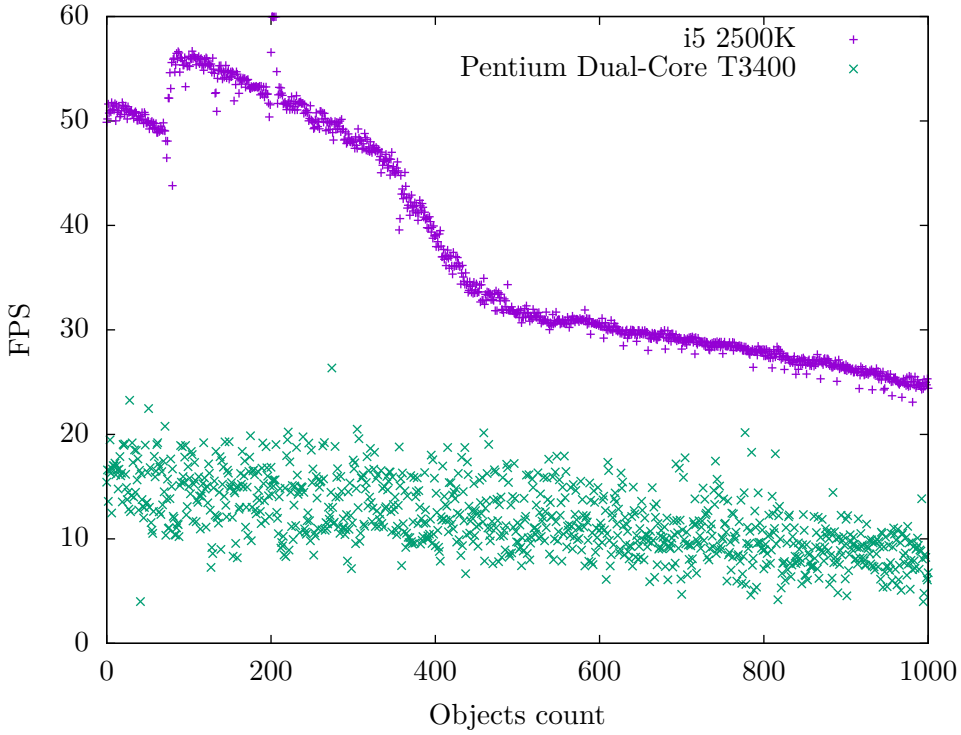

Figure 6.4: Hardware influence on the engine performance in Chrome 42.0

Since the area used in all previous tests was quite large, even larger than the display resolution of the laptop used in the previous test, a smaller area was used in an attempt to achieve a better frame-rate by the overall slowest

browser, Internet Explorer. The same scenario was used only this time with the area resolution halved in both dimensions, effectively making circa 250 tiles fill the screen instead of the original thousand. Since the amount of objects added was kept the same, at the end of the scenario, the area was four times more cluttered by objects. As Figure 6.5 shows, Internet Explorer managed twice as high a frame-rate than previously for a while, until it got overwhelmed by the sheer amount of objects added. At the end, there was no difference in frame-rate caused by the smaller size of the rendered area.

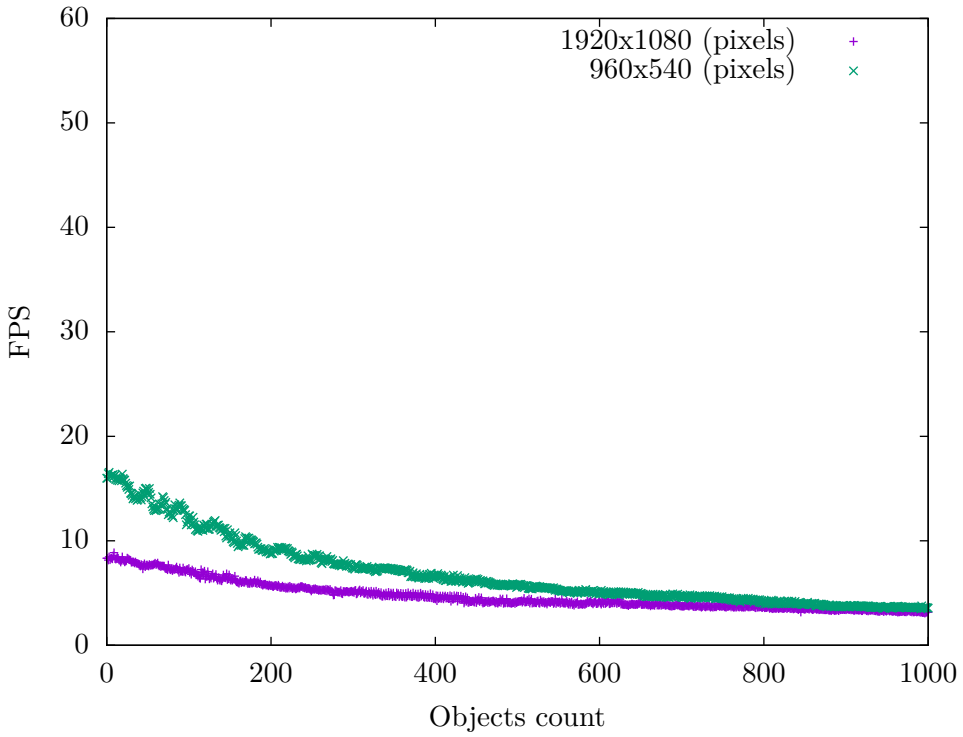

Figure 6.5: Rendered area size influence on the engine performance in Internet Explorer 11.0

6.2.4 Discussion

Over all was the engine performance rather poor, with the exception of WebKIT based browsers. On the other hand, WebKIT browsers are the most common among the users these days[53]. To create a game enjoyable in all currently available browsers, only a limited amount of tiles can be displayed

on the screen at all times. Slight improvement may be achieved by further optimization of the engine code, but according to the profiler, most of the execution time is taken up by element bounding box calculation by `getBBox` function implemented in the browsers[27]. Other large factor in the engine performance is the browser performing reflow of the layout after most of DOM modifications. In this case it might help to rewrite the engine using *React.js*, a recent library that internally creates a virtual copy of DOM and then calculates the minimum amount of real DOM changes to save time on reflow[54].

7 Conclusion

The incentive of this work was answering the question of whether the `svg` element could be used for game graphics rendering in a similar manner to the more commonly used `canvas` element. Both of these elements were introduced as a part of the HTML5 standard, but while the latter is used in many modern games, developers are discouraged from using the first one in game environments. The goal was to create an engine allowing to draw isometric graphics for any kind of tile based game. Isometric projection in games is still popular and to sufficient extent allows to display 3D scene using simple 2D rendering. For universal usage, the engine needs to strictly separate game logic and assets from the general functionality constant to any game. This namely entails isometric projection logic, including the isometric coordinate conversion[1] handling and detecting what tiles are currently in the user's window and therefore rendering nothing unnecessary. Finally, the engine needs to be able to obtain the data required to populate the screen with correct tiles and keep those up to date. That may involve only individual cell updates and the engine has to be ready for this possibility.

Even though the engine implementation attempt was successful and the result meets all requirements, wider application can be hardly recommended. The reason for this discouragement are the differences in performance of various web browsers when working with the `svg` element. As the testing showed, browsers based on WebKIT layout engine, which are currently the most widespread among web users, handle SVG in satisfactory speed, but other browsers on the same hardware manage to keep a barely playable or completely unplayable levels of frame-rate at most times. Since a developer of a web application can rarely count on the user running a certain web browser, a game built on the proposed engine would have to carefully limit the amount of displayed tiles and objects. In the view of the fact that no such strict limits are necessary when drawing the graphics to the `canvas`, it is a better option until the performance equalizes among browser platforms.

1. Map model units conversion to screen pixels and back.

Bibliography

[1] World Wide Web Consortium. *HTML5. A vocabulary and associated APIs for HTML and XHTML* [online]. 2014. [cit. 19/4/2015]. URL: <http://www.w3.org/TR/html5/>.

[2] PAGELLA, M. A. *Making Isometric Social Real-Time Games with HTML5, CSS3, and JavaScript*. Sebastopol: O'Reily Media, Inc., 2011. ISBN 978-1-449-30475-1.

[3] CHARNIAK, E. et al. *Artificial Intelligence Programming*. New York: Taylor & Francis, 2 edition, 2014. ISBN 978-13-177-6799-2.

[4] GREGORY, J. *Game Engine Architecture*. Boca Raton: A K Peters, Ltd., 2009. ISBN 978-1-4398-6526-2.

[5] CLAYPOOL, M., CLAYPOOL, K., DAMAA, F. The Effects of Frame Rate and Resolution on Users Playing First Person Shooter Games. In *Proceedings of ACM/SPIE Multimedia Computing and Networking (MMCN)*. January 18-19, 2006.

[6] MAYNARD, P. *Drawing Distinctions: The Varieties of Graphic Expression*. New York: Cornell University Press, 2005. ISBN 978-08-014-7280-0.

[7] DESAI, A. A. *Computer Graphics*. Delhi: PHI Learning, 2008. ISBN 978-81-203-3524-0.

[8] The international Arcade Museum. *Zaxxon - Videogame by Sega/Gremlin* [online]. 1995. [cit. 26/4/2015]. URL: <http://www.arcade-museum.com/game_detail.php?game_id=12757>.

[9] GABEL, T. et al. *Dune II: The Building of a Dynasty (DOS)* [online]. 1999. [cit. 10/5/2005]. URL: <http://www.mobygames.com/game/dos/dune-ii-the-building-of-a-dynasty>.

[10] Blizzard Entertainment. *Blizzard Entertainment: Classic Games* [online]. 2015. [cit. 11/5/2015]. URL: <http://us.blizzard.com/en-us/games/legacy/>.

[11] Fixaris Games. *Sid Meier's Civilization* [online]. 1996. [cit. 11/5/2015]. URL: <https://www.civilization.com/en/games/civilization-ii/>.

[12] Moby Games Contributors. *Age of Empires (Windows)* [online]. 2006. [cit. 11/5/2015]. URL: <http://www.mobygames.com/game/age-of-empires>.

[13] SAWYER, C. et al. *Transport Tycoon. History* [online]. 2013. [cit. 11/5/2015]. URL: <http://www.transporttycoon.com/history>.

[14] Blizzard Entertainment. *Blizzard Entertainment: Starcraft II* [online]. 2010. [cit. 11/5/2015]. URL: <http://us.blizzard.com/en-us/games/sc2/>.

[15] Adobe Systems Incorporated. *Adobe Flash Player* [online]. 1996. [cit. 1/5/2015]. URL: <http://www.adobe.com/software/flash/about/>.

[16] Microsoft Corporation. *Canvas Class MSDN Documentation* [online]. Microsoft Corporation, 2008. [cit. 17/4/2015]. URL: <https://msdn.microsoft.com/library/system.windows.controls.canvas.aspx>.

[17] ORACLE et al. *Class Canvas (Java Platform SE 7)* [online]. 2011. [cit. 17/4/2015]. URL: <http://docs.oracle.com/javase/7/docs/api/java/awt/Canvas.html>.

[18] Canvace. *Canvace. The HTML5 Game Platform* [online]. 2013. [cit. 11/5/2015]. URL: <http://canvace.com/>.

[19] WebCreative5. *Cangas Engine. Framework to create video games in HTML5 Canvas* [online]. 2012. [cit. 11/5/2015]. URL: <http://canvasengine.net/>.

[20] Ubiquitous Entertainment Inc. *enchant.js. A simple JavaScript framework for creating games and apps* [online]. 2011. [cit. 11/5/2015]. URL: <http://enchantjs.com/>.

[21] Irrelon Software Limited. *Isogenic Game Engine. The world's most advanced HTML5 multiplayer game engine* [online]. 2013. [cit. 11/5/2015]. URL: <http://isogenicengine.com/>.

[22] W3SCHOOLS. *HTML5 SVG* [online]. 2014. [cit. 6/5/2015]. URL: <http://www.w3schools.com/Html/html5_svg.asp>.

[23] JOFFE, B., FAUGHT, D. A., BARANOVSKIY, D. *RaphaelScape. Maze Mod with Raphaël* [online]. 2010. [cit. 13/5/2015]. URL: <http://raphaeljs.com/scape/>.

[24] Mozilla Developer Network and others. *CanvasRenderingContext2D.drawImage() - Web API Interfaces* [online]. 2014. [cit. 12/5/2015]. URL: <https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/drawImage>.

[25] CompuServe Incorporated. *Graphics Interchange Format. Version 89a* [online]. 1990. [cit. 12/5/2015]. URL: <http://www.w3.org/Graphics/GIF/spec-gif89a.txt>.

[26] Microsoft Corporation. *[MS-SVG]: The 'foreignObject' element* [online]. 2015. [cit. 1/5/2015]. URL: <https://msdn.microsoft.com/en-us/library/hh834675%28v=vs.85%29.aspx>.

[27] World Wide Web Consortium. *Scalable Vector Graphics (SVG) 1.1. Basic Data TYpes and Interfaces* [online]. 2011. [cit. 16/5/2015]. URL: <http://www.w3.org/TR/SVG11/types.html>.

[28] BARANOVSKIY, D. et al. *Raphaël—JavaScript Library* [online]. 2008. [cit. 10/5/2015]. URL: <http://raphaeljs.com/>.

[29] BARANOVSKIY, D. et al. *Snap.svg - Why Snap* [online]. 2013. [cit. 10/5/2015]. URL: <http://snapsvg.io/about/>.

[30] EISENBERG, J., BELLAMY-ROYDS, A. *SVG Essentials*. Sebastopol: O'Reilly Media, 2 edition, 2014. ISBN 978-14-919-4533-9.

[31] FIERENS, W. *svg.js - A lightweight library for manipulating and animating SVG* [online]. 2012. [cit. 10/5/2015]. URL: <svgjs.com>.

[32] BOSTOCK, M. et al. *D3.js - Data-Driven Documents* [online]. 2011. [cit. 10/5/2015]. URL: <http://d3js.org/>.

[33] BURKE, J. et al. *RequireJS. A JavaScript module loader* [online]. 2011. [cit. 10/5/2015]. URL: <http://requirejs.org/>.

[34] RESIG, J. et al. *jQuery. Write less, do more* [online]. 2006. [cit. 10/5/2015]. URL: <https://jquery.com/>.

[35] LABS, P. *Jasmine. Behavior-Driven JavaScript* [online]. 2010. [cit. 10/5/2015]. URL: <http://jasmine.github.io/>.

[36] The Apache Software Foundation. *Maven* [online]. 2002. [cit. 10/5/2015]. URL: <https://maven.apache.org/>.

[37] JíNA, V. et al. *Karma. Spectacular Test Runner for Javascript* [online]. 2012. [cit. 10/5/2015]. URL: <http://karma-runner.github.io/>.

[38] Mozilla Developer Network and others. *XMLHttpRequest - Web API Interfaces* [online]. 2005. [cit. 6/5/2015]. URL: <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>.

[39] Ecma International. *ECMAScript Language Specification*. Geneva: Ecma International, 2011.

[40] Mozilla Developer Network and others. *JSON - JavaScript* [online]. 2011. [cit. 6/5/2015]. URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON>.

[41] JBoss Community. *RESTEasy* [online]. Red Hat, Inc., 2009. [cit. 3/5/2015]. URL: <http://resteasy.jboss.org/>.

[42] FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[43] FASTERXML. *Jackson JAX-RS providers* [online]. 2009. [cit. 3/5/2015]. URL: <https://github.com/FasterXML/jackson-jaxrs-providers>.

[44] Google Inc. *App Engine - Google Cloud Platform* [online]. 2008. [cit. 10/5/2015]. URL: <https://cloud.google.com/appengine/>.

[45] Microsoft Corporation. *Array Object (JavaScript)* [online]. 2015. [cit. 13/5/2015]. URL: <https://msdn.microsoft.com/library/k4h76zbx%28v=vs.94%29.aspx>.

[46] CHEN, R. *The Old New Thing. A cache with a bad policy is another name for a memory leak* [online]. 2006. [cit. 13/5/2015]. URL: <http://blogs.msdn.com/b/oldnewthing/archive/2006/05/02/588350.aspx>.

[47] Mozilla Developer Network and others. *Node.insertBefore() - Web API Interfaces* [online]. 2005. [cit. 12/5/2015]. URL: <https://developer.mozilla.org/en-US/docs/Web/API/Node/insertBefore>.

[48] Mozilla Developer Network and others. *Window.requestAnimationFrame() - Web API Interfaces* [online]. 2010. [cit. 13/5/2015]. URL: <https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame>.

[49] ROBINSON, J., MCCORMACK, C. *Timing control for script-based animations* [online]. 2013. [cit. 14/5/2015]. URL: <http://www.w3.org/TR/animation-timing/>.

[50] KONTOS, A. *Waterfox. The fastest 64-bit browser on the web* [online]. 2011. [cit. 18/5/2015]. URL: <https://www.waterfoxproject.org/>.

[51] Vivaldi Technologies. *Vivaldi previews powerful new web browser* [online]. 2015. [cit. 18/5/2015]. URL: <https://vivaldi.com/press/releases/2015-01-27/>.

[52] Google Inc. *The Chromium Projects. Blink* [online]. 2013. [cit. 18/5/2015]. URL: <http://www.chromium.org/blink>.

[53] StatCounter. *StatCounter. GlobalStats* [online]. 2015. [cit. 16/5/2015]. URL: <http://gs.statcounter.com/>.

[54] SOBO, N. *Moving Atom To React* [online]. 2014. [cit. 16/5/2015]. URL: <http://blog.atom.io/2014/07/02/moving-atom-to-react.html>.

A Supplement contents

Source code of the proposed engine implementation described in Chapter 5 as well as the demo application showing the usage of the engine is available in a Git repository hosted on *GitHub*: https://github.com/vit-svoboda/svg-engine. A copy of the source code is also available in the thesis supplements. Key part of the provided code are the web application scripts.

B Engine user's guide

The engine provided API that makes viewport content manipulation very easy. The use of this API is demonstrated in the sample application. Part of the engine configuration are three interfaces that together provide the game specifics implementation. All of them can be implemented by a single object or each can be an object on its own. However all of the interface implementations need to be handed over to the engine during its initialization. When everything is set up, calling method `Engine.run` will launch the game update loop.

B.1 Data feed

Data describing what should be displayed on the screen are provided to the engine through an object passed as a first parameter of `Engine.init` method. When the engine needs data, the `getData` or `getDetailedData` methods are called. This data can be obtained from a remote server using HTTP requests, stored in the client memory and passed directly or in any other way suitable for the given game.

B.2 Asset management

To give the game implementation power over what is displayed in place of data discussed earlier, everything displayed is translated using an object passed as a second parameter of the `Engine.init` method. This translation is performed via methods `createTile` or `createObject` where the SVG drawing context is provided to allow proper element creation. The result is placed on the corresponding place in the viewport.

B.3 UI handling and game logic

The third object passed to the `Engine.init` method is responsible for handling user interaction. The `createUi` method is responsible for the standard UI displayed all the time, the `processDetailedData` method is a handler for detailed tile data once it is obtained and the `onClick` method is a handler for any tile or object clicked. Here most of the user interaction needs to be handled. Very little boundaries are given to other interaction implementation though.