Dynamic graphics in serious games

DYNAMIC GRAPHICS IN SERIOUS GAMES

by Timen P. Olthof

MASTER THESIS

Supervisor: prof. dr. Anton Eliëns (VU Amsterdam)
Internship supervisor: Berend Weij, MA (Mijn naam is Haas)
Second reader: dr. Željko Obrenović (TU Eindhoven)

CONTENTS

PREFACE

Computer scientists are system builders. As systems become more complex, computer scientists will have to become more complex too. During my study Computer Science at the Vrije Universiteit Amsterdam, I have learned to analyze, design and build ever more complex systems and use increasingly complex technologies.

Users are busy people. They want systems that are easy and simple to use, yet perform the most complex tasks in a second and fit within their ever more complex daily lives. Over the last decade I have seen users struggling more and more while using new technologies.

The connections between humans and computers often remain difficult and unintuitive. The only solution to this problem is education, and it is working. I am very happy and proud to have done an internship at a company that is on the forefront of computer aided education, and of which the product combines the essential ingredients of modern learning: solid teaching material, explorative narrative and a fun experience.

The last few years I have specialized myself in the field of multimedia, working on rich internet applications, pre-

sentation and visualization applications, generative art, content distribution and ebooks, computer graphics, multi-touch applications and mobile applications. I believe that in these times, even now that content may often appear to be obscured by form, the message is also the core of the medium and is still the ultimate connection between humans and machines. Of course there are a lot of media and, consequently, a lot of connections.

I would like to thank prof. dr. Anton Eliëns for his excellent guidance during my entire master study. Any student would wish for a supervising professor like Anton. I also would like to thank Berend Weij for his great supervision of my internship, and Peter Peerdeman for being my most direct critic and friend during the whole internship and master study.

CHAPTER 1

INTRODUCTION

1.1 Multimedia

Multimedia is a badly defined field of expertise. It includes such diverse topics as document engineering, content creation and annotation, content distribution, interactive applications and experiences, serious games, social media, visual design and human computer interaction. And there are many 'media'. Different types of sound, images, and tactile interfaces, and different types of ways for humans to communicate and interact with (things in) the world. In fact, the concept of an interface is possibly the most important concept of multimedia. Media are by definition 'in between' a sender and a receiver (although it may not be clear who is the sender and who is the receiver). Media are connectors, and multimedia deals with connecting many things. The field of multimedia is 'artistic' in the sense that it requires both technique (technical skill) and talent (inspiration). It deals with education of skills and techniques, as well as harnessing the creativity of humanity.

1.2 Serious games

Media are very hard to distinguish from the message(s). According to Marshall McLuhan, games are to humans as social beings what technology is to humans as animal organisms ([49], p255). In other words: where technology supports and extends tasks that have to do with everyday life (eating, moving, working), games support and extend tasks that are related to the social part of life (communication, playing, performing). Of course, these parts of life are heavily 'intertwingled'.

Serious games are games that interwingle entertainment and education. The idea behind this is that the combination of entertainment and education improves learning, not only because it evokes curiosity, but also because it

offers control to the player and a certain challenge within a given context [20]. Serious games give direct feedback about learning accomplishments, offer a relatively realistic sandbox for experiments and insert learning goals into concrete situations [37].

1.3 Mijn naam is Haas

Mijn naam is Haas (My name is Haas) is a games production company with the serious game *Mijn naam is Haas* as their core product. Until now, three CD-ROM games for Mac and PC as well as two picture books have been released and more products are on their way. The products are targeted at children aged 3-7, and help the children to improve their dutch language skills in a creative and playful way.

One of the unique points of the Mijn naam is Haas games is the fact that the game commences with the instruction for the player to draw the world of Haas, the main character of the game. After the player has drawn a horizon line, a metaphor for the story line, the world of Haas is generated upon it and an interactive, non linear narrative unfolds as Haas accomplishes his journey through the game world. During his quest, Haas and the player encounter a number of problems that have to be solved to continue, and while solving these problems, the player is brought into contact with the dutch vocabulary in a didactically responsible way. Solving the problems is, again, done by drawing objects in the game world. Those objects will then influence the game world so that the problem is solved and Haas can continue his journey.

The Mijn naam is Haas games are developed in the ActionScript3 (AS3) programming language, which compiles to a Flash (SWF) application. Flash is currently the de facto standard for (rich) internet applications. The Mijn naam is Haas game is currently developed as an Adobe

Integrated Runtime (AIR) application, but the goal is to distribute future versions as a web service application running in the browser. This is easy to accomplish, because ActionScript can be compiled to either a standalone AIR application or a Web application. In chapter 3, I will discuss the Flash platform in more detail.

1.4 Game Design

Computer game design is an innovative sub discipline of the field of multimedia. In some sense computer games are the newest type of media, succeeding film and television. While games are becoming more and more realistic, reality becomes more and more like gaming. Games can be considered art, in the sense that they reflect and represent the human condition.

Learning and playing have always been intrinsically related, as all children and young animals learn things by playing. It is only logical that learning and playing are moving into the digital age, like many other parts of human life. As this is not a text about learning, we won't go into pedagogical discussions about what constitutes true learning, but at least a development can be identified towards more creative types of education.

Creative learning transcends the classical educational distinction between talent and technique, where a talent is an innate mastery of a certain skill and a technique the execution of a set of rules that can be learned from a teacher. Instead choosing one of the classical sides, creative learning focuses on placing student users in *educational situations*, where students have to solve a certain problem with a certain toolset. As a result, students gain *experience*, and improve themselves. In (serious) games, experiences are structured similarly.

A phenomenology of game design

A game starts out with a certain individual *situation*, an opportunity of the user or player to do something. This situation can for example be the detection of a certain pattern in what was previously apparent chaos, a 'cognitive dissonance' or an aesthetic peculiarity. Often, this situation has some external meaning or reference to reality. The situation can also be a metaphor for an event in real life.

But a situation is never alone. It is surrounded by preceding and following situations, by situations that occur at a slightly different place, and, thanks to the fantasy of the human mind, by situations that are not actualized and may never be actualized. In other words, the situation is integrated within a spatio-temporal *world*. In most games, the notion of a game world is mainly one of topology and projection. The game world is shown to the player as a 2D (top or side view), 3D (isometric or frustum based), virtual reality or immersive projection. But more importantly, it is shown as a world that can be explored, navigated, mapped and possibly expanded.

For the user to be able to be in a situation implies that the user is able to *act* in a situation, which means to not just passively observe, but to 'actually' do something. Doing something requires the skill to do something. Doing without skill is doing anything, not something. Skill can for example be aiming, timing, concentration, strategy allocation, agility, forcing, moving or another activity, and associated with skills can be affordances and features like size, speed, power, strength, memory, knowledge, territory, intuition, etcetera. Within computer games, skills are usually enforced with the help of game mechanics, user interface design and virtuality (immersion). If the player can leave the game, his freedom is absolute. If he cannot, his freedom is control within a situation. Most computer games can be left. Life cannot be left.

Activity implies development, and a player will gain *experience* with every act. Experience may come in the form of power, superpower(s), magic, evolution, progress, rewards or increasing skill level(s). Acts are executed with a certain goal or objective in mind, which may or may not be accomplished. It is important to note that experience is not just gained when goals are successfully accomplished, but that failing also brings experience, a fact that is often forgotten. The best objectives are those that are not too easy and not too difficult for the player, and in serious games, the difficulty of objectives can often be matched to individual players. Experience is the primary learning component of games, and the most important educational part of game experience is feedback such as failure costs and success rewards. Of course, fairness is important here, for a game that unjustly punishes players will be tossed aside quickly.

Interactive narratives

Narrative is the link between the absolute and the concrete, as narrations and stories are registrations of series of concrete events, but at the same time contain absolute situations that reoccur in new concrete situations again and again. Narratives compare abstract acts (ethics) with concrete acts (praxis). It is no surprise that narratives are often about good versus evil.

But when reading, listening to or watching a story, the audience can not really act itself. Consequently, classic narratives are more about ethics than about praxis. Interactive narrative changes this, by adding 'actual' situations to storyline. In a sense, movies and books are (just) worlds, while interactive narratives are worlds with situations. While experience from watching movies is only experience with actions of other people and characters, experience from interactive narratives and games is experience with

personal responsibility.

It is important to align the responsibility of the player and the game character (avatar), and that the personal goals of game characters are realistic even if the goals are to be achieved by the player. The goals should be justified by the (supposed or projected) internal motivation of the game character, but also become the personal goals of the player. This is an absolute requirement for creating 'involvement' of the player and the game character.

CHAPTER 2

RESEARCH QUESTIONS

2.1 Introduction

In this chapter I will discuss the research questions of my internship at Mijn naam is Haas. The general research problem for this master project internship ultimately rises from the wish to keep improving the unique Mijn naam is Haas game experience. While the world of Haas is already a very elegant and coherently styled world, the game is under constant development to improve the freedom and subtlety of the world of Haas as experienced by the player. This means the research question can be formulated as follows:

> (How) can the (visual) game experience of the *Mijn naam is Haas* game be improved?

In this research question, a few different tracks meet. First and foremost there is the element of the coherence and diversity of the visual style of the Mijn naam is Haas game. With the use of dynamic graphics, the (more or less) subtle run time manipulation of the game graphics, it is possible make the world of Haas more organic. At the same time, the visual style of the game needs to remain coherent. In chapter 4 I present the results of this part of the internship.

Secondly, there is the more technical track of improving the game experience by improving the functionality and performance of the render engine. If the game renders faster, it will feel more direct, and/or there will be (more) cpu time left to add more (complex) visuals. The default `flash.display` renderer is very powerful but it is difficult to optimize and extend because it is implemented inside the FlashVM virtual machine, not in ActionScript3 code. The limitations of the `flash.display` renderer can perhaps be circumvented with the design and construction of a custom render engine. Solutions to this part of the research question are listed in chapter 5.

Of course these research tracks are joined in the task of translating the results into functional designs for the Mijn naam is Haas game, and implementing these designs into actual software components.

2.2 Research questions

In summary, the following three sub questions can be distinguished:

1. (How) can the visual style of the world of Haas be enriched with the use of dynamic graphics? Which approaches are possible and how do they compare in terms of effectiveness and performance?
2. How do different types of render engines compare in terms of functionality and performance? How do they compare to the default FlashVM render engine?
3. Within the Mijn naam is Haas game, which approaches to improve the game experience with the help of dynamic graphics are most effective? How could these approaches be designed and implemented? (How) can a custom built sprite based render engine improve the (visual) game experience, and how could such an engine be designed and implemented?

It is clear that these sub questions are closely interconnected, and as such require a holistic approach to be solved. The solution of these questions also profits from an explorative and experimental hands-on approach.

A number of problems and requirements can be distinguished in the context of the research question and sub questions. For clarity, these problems and requirements

have been divided into two sections, one having to do with dynamic visuals and another one dealing with render engine performance.

2.3 Dynamic visuals

The Mijn naam is Haas game has a unique and consistent visual style, which is intelligently designed instead of pseudo-randomly generated, and as such has a coherent look and feel. When the visual elements of the game are made more dynamic, it is very important that the overall visual style remains intact and coherent. This means adding dynamics to the graphics should be done in such a way that the visual elements remain within the boundaries of the overall visual style, adding richness to the visual representation of the story universe while sustaining its credibility and believability. Because players are allowed to add game world objects to the screen by drawing anywhere on the game screen, a relatively big part of the landscape composition is in the hands of the player instead of in the hands of the designer(s). This introduces the risk of the landscape becoming too cluttered and imbalanced. This loss of control over the composition is partly recovered because the designer remains in control of the objects that are actually added based upon interpretation of the user's drawing actions, but a second method to do this is the distribution of game world objects over multiple visual layers. This will not only give the world more depth, but also offers the solution of putting game world objects in back layers when the screen is about to become to cluttered. However, users should not be tempted to actually try and draw in the background, because this may create a counterintuitive drawing experience. Instead, the visual style should make it clear to the user what the appropriate drawing area is.

Often, multiple game objects are in a certain relation to each other. One of those relations is that of one object existing on, in or at another object; the first object is a *sub-object* of the second. This is for example the case when the character Haas carries a tool (for example an umbrella) or when living objects grow in or on other objects (for example berries on a bush or ivy and fungi on tree stems). Sub-objects can be rendered onto their parent object dynamically. If however these sub-objects have some kind of movement/animation, they need to be rendered as separate visuals / world objects.

Dynamic graphics can be used to reflect global atmosphere, weather or mood changes, but also to provide visual diversity between different visual elements and create a more organic world.

To accomplish dynamic visuals, two rendering methods are available. One method is to create a pool of different pre-rendered visual representations of the same game world object. Another method is to generate the visual representation at runtime based on a list of parameters. However, both of these methods have (dis)advantages: pre-rendering visuals only leads to a small, limited degree of dynamics, but there is much control over the resulting graphics; generating graphics at runtime has the advantage of not having to design all graphics beforehand, but it may lead to too much diversity and an inconsistent visual style. In order minimize the disadvantages, a combination of the two methods can be used, for example a method in which the visuals are not completely randomly generated (or composited) at run time, but instead from a limited set of predefined input elements and values. This is effectively a balance between randomization and control.

Three main approaches for dynamic graphics can be distinguished:

1. shape variation (compositing): This approach, dynamic composition, probably helps most to maintain the Mijn

naam is Haas visual style. It works by building a visual representation from a collection of partial visual elements. For example a tree can be built up by selecting a canopy from a pool of canopies, a trunk from an array of trunks and for example a root from a list of roots and optionally branches from a selection of predefined branches. While all partial elements are pre-rendered, the whole tree is not. This results in an exponentially growing number of different visual representations, that nonetheless all comply with the global visual style. In addition to this, different partial elements can be distributed across multiple visual layers, giving more depth to the visuals.

2. size variation (scaling/skewing): Another approach is scaling, making pre-rendered visual elements bigger or smaller, thereby creating a variance in size. However, all visual elements in the Mijn naam is Haas game are carefully designed in proportion to each other, so that they fit together well. As scaling would lead to differences in proportions between different visual objects, it is not the best approach to create a more dynamic visual style. Skewing (scaling without preserving the aspect ratio of visuals) is also not a good idea, since it will lead to ugly visuals that don't fit in with the overall Mijn naam is Haas visual style. Additionally, variations in size do not effectively create the idea of richness in visual elements, but rather expose the limits of the visual style.

3. color variation (color palette(s)): Finally, color variation is a powerful approach to generating dynamic graphics. The idea here is not to use random color variations, but maintain a certain degree of control over the visual style by combining pre-rendered visuals with predefined color palette(s). This can be done

with conventional color replacement algorithms, but more sophisticated custom algorithms can of course also be used. Dynamic color palettes for the visual elements can play an essential part in controlling the overall atmosphere, because differences in seasons, time of day, weather types, mood settings, etc. are mostly established by altering the global color palette (on a per object basis).

Associated with dynamic visuals are a number of workflow optimizations, such as visual editors for managing, adapting and possibly converting the game graphics after they are created in a graphics editor. One could think for example of a Photoshop plugin that exports to a file format usable for the Mijn naam is Haas game, a landscape coloring editor or a 'visuals test lab' to create and generate color palettes or test partial visual element compositions. During my internship, I chose to build a separate visual assets management application.

2.4 Render engine performance

The default `flash.display` renderer is in general very fast (because it is implemented natively in the FlashVM), and very powerful, because as a programmer you only have to add visual elements to the display list, and the renderer takes care of everything else. But there are also a many problems with the default Flash renderer:

1. As the default Flash render engine is not specifically designed for the Mijn naam is Haas game, it sometimes does too much work, for example re-rendering game visuals or re-caching. A custom render engine based on a custom FRAME notification could improve performance and thereby the game experience, by focussing on those points that are specifically important for the Mijn naam is Haas game. In this case all

render tasks like positioning, compositing and drawing have to be customly implemented.

2. The default Flash `BitmapData` class has a size limitation of 16,777,215 pixels. Because the current render engine uses the zoom functionality that is integrated in the `flash.display` renderer to zoom to a specific region on a large bitmap, the maximum game world size is effectively limited to about 4095x4095 pixels. A custom render engine that doesn't use zooming, but instead only renders the user's current viewport could transcend this maximum world size limit. Unfortunately, this also requires appropriate changes in the Model side of the game (for example to accommodate terrain type checks in a different way).

Currently, visual game elements (visual assets), are loaded from a few big compiled Flash movies (libraries). Internally the different images (a lot of them animation frames for different 'postures' of different game objects) are stored as lossless compressed PNG image files. To use these assets in a custom render engine, they have to be converted (drawn) to a BitmapData object. This takes time. Luckily there are a number of possible optimizations to this method.

1. It can be faster to load and convert larger images than smaller ones. By using *sprite sheets* (larger image files containing multiple animation frames of the same visual object), loading and rendering can potentially be sped up.

2. Another method is to store the visual data in a custom format that can be loaded to a `BitmapData` object. This can be done by serializing visual assets to `ByteArray`s and storing this as a binary file. Loading from a binary file is much faster than loading separate image files like PNG's.

3. All serialized image files can be combined into one ore more binary assets files to potentially speed up loading and rendering even further.

4. ActionScript 3 provides native `zlib` compression for `ByteArray` objects (which is probably the same compression as is used internally by the Flash SWF libraries to provide lossless PNG compression). This compression functionality can be used on a per frame, per animation (posture), per object basis or on the complete asset library as a whole.

CHAPTER 3

TECHNOLOGIES

3.1 Introduction

In this chapter I discuss the tools and technologies related to dynamic graphics in the Mijn naam is Haas game. This chapter not only provides the technical context in which the internship research has been done, but also gives an overview of relevant technical background research into technologies that support or could support the solution of the challenges the research problem of this internship poses. Successively, the Adobe Flash platform (ActionScript, Flex, etc.), Adobe Pixel Bender, Adobe Alchemy, HaXe, PureMVC and general programming optimizations are discussed.

3.2 Flash and ActionScript

The Adobe Flash platform has been the dominant and de factor standard for rich internet applications for years now. Almost every web application that is more complex than a web form is built in Flash, and especially animation based web applications such as games can be built very well with Flash, originally an animation platform.

The Flash platform is based around the Flash Player, a piece of plugin software for browsers and mobile devices that contains the Flash virtual machine. The FlashVM interprets compiled .swf files that contain Flash byte code. This means that Flash applications are virtually operating system independent. Flash players are available for all major operating systems.

In recent years a lot of new competitors have joined the market of rich internet application platforms, but to date none of them deliver the universal (multi platform) executability, easy development process and feature richness of the Flash platform.

Probably the biggest competitor is HTML5/JavaScript, but that technology still suffers from a lack of standardization: every browser client requires specific implemen-

tations of the rich internet application, and no universal standards exist for video codecs, animation and interactivity.

Microsoft SilverLight and Unity are more standardized, but require additional plugins. Of course the Flash platform also requires a plugin, but the installed base of the Silverlight and Unity plugins is nowhere near the 97-99% penetration rate of the Flash Player. Additionally, the Flash plugin will be integrated into the upcoming versions of the Google Chrome browser by default, so no separate installation will be required.

The most important development in web technologies is the movement of applications to mobile devices such as mobile phones and tablets. In response to this development, Adobe has created packagers for both Apple iOS and Android operating systems to allow Flash applications to be deployed to those devices. While these packagers are still in their infancy, they present an easy way of targeting multiple problems with one codebase.

The Flex framework is a set of quickly usable GUI Flash components released by Adobe to make the construction of rich internet applications easier. Adobe is working hard to optimize the Flex framework also for mobile applications.

Another development is the AIR runtime (Adobe Integrated Runtime) which allows Flash applications to run as desktop applications, and abstracts system services to Flash applications more directly. A mobile version of the AIR runtime is also being developed by Adobe.

The main advantage of the Flash platform is the fact that it is really a platform independent framework. There are small differences between implementations on different platforms, but these concern mainly performance issues and platform specific functionality (such as GPS devices in mobile devices, or touch screens). From the point of view of the programmer, any Flash application will 'just work' on each computer that has the Flash player installed,

which is almost every computer. Additionally the Flash platform's work flow can be as easy or complex as is required by each specific project. Flash works just as well with very easy rich internet applications or mini games as with highly complex multicomponent application architectures.

Disadvantages of Flash are that its development (compiler and languages) are controlled exclusively by Adobe, which means that as a user of the framework (a programmer) you are dependent on strategic decisions made by Adobe. Unfortunately, Adobe has not taken compiler optimizations very seriously. Hardware acceleration has been introduced in Flash Player 10.1, but only for very specific (narrow) situations, like h264 video and mobile applications [54]. Another disadvantage is the fact that de development of the Flash platform has a long history, and that all versions are backwards compatible. This makes the platform more sluggish than necessary.

The Adobe Flex SDK

The Adobe Flex SDK is a collection of required and optional compilers and tools to compile Flash, Flex and AIR applications. Despite being named *Flex* SDK, it can also be used to compile and package non-Flex (pure ActionScript3) projects. It is structured as follows:

/ The root folder of the SDK contains a set of read me files and license description files.

ant/ contains the files needed for the Ant builder tool, a Java-based build tool. It is used (and not *make* for example) because it is cross platform, and uses easy configurable *xml*based build configuration files.

asdoc/ contains support files for the *asdoc* application. The *asdoc* application is used to automatically gen-

erate documentation files from ActionScript applications.

bin/ provides a cross-platform abstraction (Windows, Unix, Linux, MacOS) for the compilers and tools. It does this by providing executables (shell scripts and batch files) that call the actual platform independent java application files (which are located in `lib/`). This way, from any platform the aliases in bin/ can be executed, which in turn will launch the platform independent jar application files from `lib/`.

frameworks/ contains support files for the compilers and tools, like Ant build configuration files for the different targets: regular flash and AIR, and some other support resources.

lib/ contains the actual compilers and support tools (java applications). In principle they are not executed directly, but are called by the aliases in `bin/`.

runtimes/ contains the different runtime executables (Flash Player, Adobe AIR runtime) for multiple platforms (win/linux/mac).

samples/ contains example code.

templates/ contains packaging templates for Flash web and standalone AIR applications.

While the Adobe Flex SDK is a very powerful collection of tools (henceforth referred to as *the Flex compiler*), there are a few problems with it. Firstly, the Flex compiler is not that well optimized as other modern compilers (such as for example the *gcc* c compiler and c++ compilers are). As a result, the Flex SDK compiler(s) don't produce fully optimized applications.

Secondly, the Flash Virtual Machine by default only uses one processor core, and application code doesn't run

multithreaded. Since modern processors often have multiple cores and/or multithreading optimizations, Flash applications could run a lot faster if they could make use of multiple cores and or multithreading.

Thirdly, Flash applications do not make use of hardware acceleration. Hardware acceleration is the use of specialized hardware components to perform specialized tasks faster than they could be performed when they are implemented in software running on the main CPU(s), and consequently Flash applications don't use the GPU to speed up graphics related tasks.

Finally, because all Flash applications run in a Virtual Machine, there is an extra layer of interpretation between the application code and the processor. This layer provides security and stability, but also leads to slower execution than native processor execution. Memory access is also abstracted, causing speed loss as well.

There are number of strategies that can be applied to create faster applications. Of course there are general ActionScript performance coding optimizations (just writing more efficient code), but there are also a few extensions of and alternatives to the default Flex compiler workflow. Adobe Pixel Bender and Adobe Alchemy are official research projects by Adobe, while HaXe is an open source project.

3.3 Pixel Bender

Adobe Pixel Bender is a cross platform and cross application (it also works in Adobe Photoshop and Adobe After Effects) pixel shader programming language. Pixel shaders are small applications that perform pixel color calculations in a highly parallelizable way, that are generally executed on the graphics processing unit (GPU). With Pixel Bender, these shaders can in principle be executed by the CPU (cpu mode) and by the GPU (gpu mode). Unfortunately, the

Flash VM only supports cpu mode.

To create a Pixel Bender *kernel*, as a Pixel Bender code fragment is called, Adobe supplies the free Pixel Bender Toolkit. In this application, available for Mac and Windows, these kernels can be programmed using a kind of XML/C like programming language. A kernel can than be tested on one or multiple images in the Pixel Bender Toolkit application, and can be exported for use in Photoshop or Flash.

Pixel Bender kernels usually take one ore more input images as input and result in one output image. The kernel is executed on each pixel of the destination image. An example Pixel Bender kernel (that just outputs the pixel at the coordinates of the output pixel at the current coordinate) is listed below:

```
<languageVersion : 1.0;>

kernel HaasFilter
< namespace : "com.mijnnaamishaas.pixelbender";
  vendor : "Mijn naam is Haas";
  version : 1;
  description : "Haas Bender Test";
>
{
  input image4 src;
  output pixel4 dst;

  void evaluatePixel()
  {
    dst = sampleNearest(src,outCoord());
  }
}
```

In summary, Pixel Bender can be used in five ways:

1. Filter: the kernel is applied as a filter for a DisplayObject. To do this, the pixel bender kernel is loaded into a ShaderFilter object, which in turn is added to the DisplayObject's filters array (the `filters` property of a DisplayObject). The ShaderFilter object can

also be manually applied using the `applyFilter()` function.

2. ShaderFill: the kernel is used to generate a fill for a Flex component. This is done with the help of the `Graphics.beginShaderFill()` method.

3. Blend: the kernel is executed to calculate the blending of two images or visuals. This is done by setting the `DisplayObject.blendShader` property.

4. ShaderJob mode (asynchronous): A kernel can be executed by adding an eventListener to the shader and starting it with `ShaderJob.start(false)`. This tells Flash not to wait for completion of the ShaderJob. Instead, as soon as the job finishes, it will dispatch a ShaderEvent to the eventListener. The main program continues while the job is being executed, and the job itself is run on a different core or thread, if available. The ShaderJob method executes a bit faster than the other (synchronous, blocking) methods, but it takes some time to start the ShaderJob and there is some overhead for the eventListener. The ShaderJob also has to be restarted each time it is used again.

5. ShaderJob mode (synchronous): this can be done by starting a ShaderJob with the waitForCompletion parameter set to true (`ShaderJob.start(true)`). The Flash Player will block until the ShaderJob has completed, and afterwards continue the main program.

Pixel Bender can not only be used for (shader) graphics, but it can also be (ab)used to execute large sets of mathematical calculations tasks in a faster way. One way to do this is by 'transforming' (reformulating) a mathematical problem in graphical terms.

A problem with Pixel Bender is that its speed is very much dependent on the platform the application is executed on, even more so because Pixel Bender doesn't run on the GPU in Flash (yet). The Filter, ShaderFill and Blend modes run multithreaded and will scale across multiple CPU cores, but currently the only way of using multiple cores/threads by yourself is executing an asynchronous ShaderJob. This will cause Flash to execute the complete Shader on a separate core or in it's own thread, but even then, this is only efficient with Jobs that are large enough to overcome the overhead of setting up a-synchronization. Additionally, a number of performance optimizations can be made to allow faster execution [44]:

- Using 3 channels (no transparency) is faster than 4 channels.
- Use pre-calculated constants.
- Only process the relevant part of the BitmapData.
- A ShaderJob is faster than ApplyFilter
- Processing BitmapData is faster than using ByteArrays. Using Vector<Number> is even slower.
- Try to avoid conditionals by adapting the kernel flow.
- For smaller datasets, synchronous ShaderJob is faster than an asynchronous one.
- Don't use input data structures to store output data.

3.4 Alchemy

Adobe Alchemy is an official Adobe project that makes it possible to compile C and C++ source code to Flash byte code that can be executed in the Flash Player. This is accomplished with the help of the *Low Level Virtual Machine*

(LLVM) compiler infrastructure. The LLVM framework is a compiler framework that allows the use of various *front-ends* to compile code written in various programming languages (including C(++), Java, ActionScript and others) to LLVM intermediate code. Next, the LLVM offers multiple back-ends to compile this intermediate code to machine specific machine code. In the Alchemy project, Adobe created a back-end for the LLVM framework that compiles intermediate code to FlashVM machine code, effectively allowing all languages that are supported by LLVM to be compiled to FlashVM byte code.

This not only makes it possible to reuse all those little C libraries that are available on the web within Flash applications, but in fact in many cases also produces faster Flash byte code than the default Flex SDK compiler. This is possible, because the LLVM compiler uses a few extra instruction codes that are available in the FlashVM, but (strangely enough) are not used by the default Flex compiler. There are other reasons why Alchemy may produce faster applications than regular ActionScript. The Alchemy compiler framework supports *inlining* of instructions, which means that individual assembly codes that would take a function call to execute in regular ActionScript3 are inserted directly into the instruction stream. Because function calls are very expensive operations, inlining leads to a big performance improvement, especially on small operations that are executed often. The third reason behind Alchemy's better performance is the fact that the LLVM compiler framework does a lot of optimizations while converting the C code to ActionScript, while the regular ActionScript3 compiler does few to no optimizations. Finally, because Alchemy generated applications run in fast memory themselves, and use faster memory to store data, execution performs faster [33], [34].

The use of alchemy requires an adaption of the programming workflow. Generally, an application will con-

sist of an ActionScript3 part that makes use of a second part of code written in C and compiled with Alchemy. The C/Alchemy part is to be programmed as a C application that defines a code library with library routines. This library is than compiled with the Alchemy compiler to create a .swc library file that behaves just like a regular .swc library, and can thus be used in any ActionScript3 project. This .swc file will be a bit bigger than most .swc files because it contains POSIX overhead for all relevant C calls. An example of an Alchemy C code application is:

```
#include "AS3.h"

int i;

AS3_Val init(void* self, AS3_Val args)
{
  i = 0;
  return 0;
}

AS3_Val increment(void* self, AS3_Val args)
{
  i++
  return i;
}

int main()
{
  AS3_Val initMethod = AS3_Function(NULL, init);
  AS3_Val incrementMethod
    = AS3_Function(NULL, increment);

  AS3_Val result
    = AS3_Object("init: AS3ValType,increment: AS3ValType",
      initMethod, incrementMethod);

  AS3_Release(initMethod);
  AS3_Release(incrementMethod);

  AS3_LibInit(result);

  return 0;
}
```

This code will expose two functions, an `init()` and an `increment()` function to the ActionScript programmer who uses the resulting swc file. Within a Flash application, the following ActionScript 3 code is used to open and use the swc:

```
package
{
  import cmodule.AlchemyExampleLib.CLibInit;
  import flash.events.Event;

  public class AlchemyExample extends Sprite
  {
    protected const lib:Object
      = (new CLibInit()).init();
    private var counter:int;

    public function AlchemyExample()
    {
      //initialize counter
      counter = lib.init();

      addEventListener(Event.ENTER_FRAME,
        enterFrame);
    }

    protected function enterFrame(e:Event):void
    {
      //increment counter every frame
      counter = lib.increment();
    }
  }
}
```

Considering the complexity of the process, this works actually relatively easy, but there are of course also a few downsides of using Alchemy. One problem is that in the code, communication between the AS3 part and the C part is relatively expensive (performance wise) to do. Luckily there are a few 'tricks' to quickly move data both ways [74], [64], but still it is best to prevent as much communication as possible (switching back and forth between contexts is called "marshaling" [32]). A similar problem occurs also

during the actual coding of the application; as the C part requires a separate workflow, the programmer finds himself switching back and forth between the Flash and C workflows, which also has some overhead costs.

Is Alchemy in practice really faster than ActionScript3? Theoretically yes, but the practical results seem to vary. There are a lot of examples where Alchemy seems to speed up tasks very much, for example with JPEG encoding [42], [39], and particle system rasterization [35], but other experiments suggest that it may be very difficult to achieve big speedups in real world applications [7], because of the overhead Alchemy generates and the general difficulty to fit Alchemy code into an ActionScript3 application.

3.5 HaXe

HaXe is an open source programming language and compiler that can target multiple platforms including the Flash Virtual Machine. It was created by Nicolas Cannasse, who is also the main developer. HaXe's syntax is much like the ActionScript3 syntax, and quite some pieces of code can be copied between HaXe and ActionScript3 with little to no changes. The HaXe compiler supports compilation of HaXe code to multiple platforms, including Flash, JavaScript, Neko, PHP and C [36].

HaXe is much like Alchemy in that the HaXe compiler supports the fast virtual machine opcodes that the default Flash compiler doesn't use, supports inline assembly, and contains more optimizations than the default Flash compiler. Just like with Alchemy, HaXe programs can be compiled to a reusable .swc library file that can be included and used in any regular ActionScript3 application [68]. However, HaXe compiles directly to Flash code, without using LLVM as an intermediate stage, and a lot of classes can be compiled exactly into their as3 counterparts. Also, because HaXe doesn't need to support the whole C POSIX

standard, the .swc overhead is much smaller.

Consider the following example HaXe code, that simply draws a box:

```
class Test {
  static function main() {
    var mc : flash.MovieClip = flash.Lib.current;
    mc.beginFill(0xFF0000);
    mc.moveTo(50,50);
    mc.lineTo(100,50);
    mc.lineTo(100,100);
    mc.lineTo(50,100);
    mc.endFill();
  }
}
```

Here, the main downside of HaXe becomes apparent: that it is *somewhat* like ActionScript3, but not *completely* like ActionScript, which might be difficult to switch to as a programmer. Another problem is that the project is not officially supported by Adobe, so there is not really a guarantee that the project will be kept up to date. Until now however, it is being maintained very well. The compiler seems to be very good, and there are quite a few examples of well working HaXe applications, for example web games [30].

3.6 PureMVC

As discussed before, Flash supports applications of every complexity. For larger applications, *design patterns* are usually used to make the large source code repositories more readable and better organized. An often used design pattern is the model view controller (MVC) design pattern. PureMVC is a popular model view controller code library, and an ActionScript 3 version of PureMVC is available [58].

The model view controller design pattern divides all software components into three categories: models (com-

ponents that manage data), views (components that show information to the user and offer interaction to the user), and controllers (components that contain business logic to perform operations on data and update views). The goal of this approach is to decouple data storage from the way data is shown. This leads to better data structures, cleaner view components, and makes expansion of the application easier.

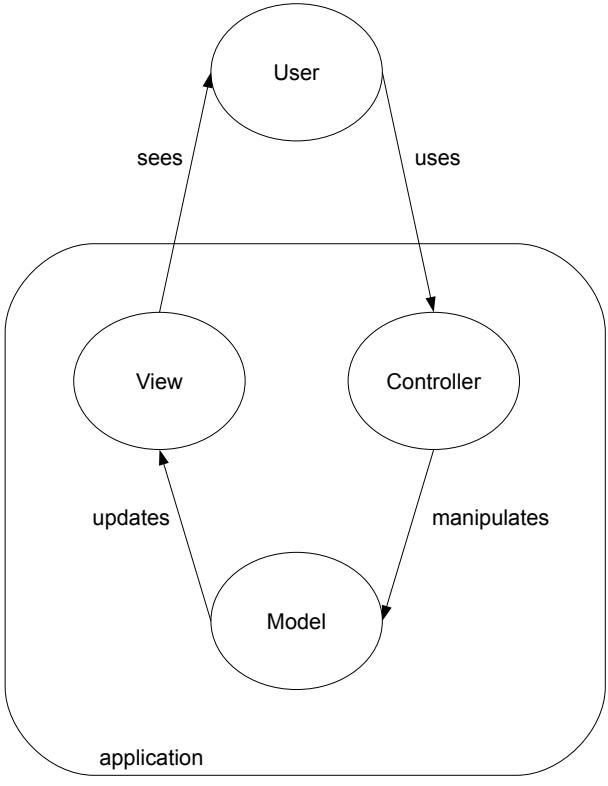

Figure 3.1: The basic Model View Controller pattern

PureMVC slightly adapts the basic model view controller design pattern. The central component is the singleton Facade, which functions as the central hub through which all components can access the components they are allowed to communicate to. Within PureMVC the Model is an abstract concept that exists in the form of one or more Proxies that manage data objects, the View is an abstract concept that exists as a set of Mediators that each manage a UI view component, and the Controller is an abstract concept that exists in the form of multiple Commands.

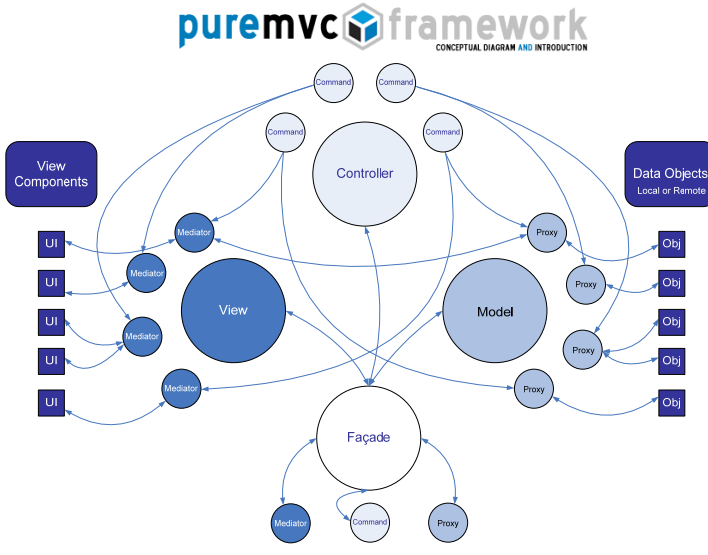

Figure 3.2: The PureMVC design pattern architecture

Because the Mijn naam is Haas game is using the PureMVC framework to structure the software components architecture, I also used it in this internship project. It is a very powerful framework and forces a programmer to write well structured components.

3.7 Other optimizations

Some other ActionScript3 optimization strategies include:

- Post compile *optimization* using Apparat. Apparat is "a framework to optimize ABC, SWC and SWF files." [17]. It can be used to speed up and compress compiled Flash applications and libraries. Apparat does this by taking already compiled Actionscript ByteCode (ABC) and speeding up this ABC by applying inlining of short methods, resulting in less instructions in the application and thus a faster application [18].
- Choose the right *data structures*. ActionScript3 supports (among others) BitmapData, Vector and ByteArray data structures. In different situations, different data structures are faster, so it is important to choose the right ones for the right situation [51], [72].
- Do less work. Usually when processing a large set of objects, a lot of useless work can be avoided using *culling* strategies (determining at runtime which objects are not relevant for the current context, separating them from the objects that are relevant, and finally processing only those objects that are relevant). For example, when building a render engine, Visuals that are not inside the current viewport, or visuals that are invisible for other reasons, don't need to be rendered at all. Techniques like this are used very heavily or example in 3d render engines [38]. Another strategy to prevent doing useless work is *caching*, in which objects that have not changed since the last iteration of a repeated process are not processed again; instead, the previously processed version is reused.
- Efficient code. The most obvious way to optimize ActionScript3 applications is by writing efficient code,

for example using different loop types (for, while, for each, etc.), instantiating and scoping variables with care, reuse variable, using bit/byte operations like bit shifts and logical operations like AND and OR, and smart typing and casting of variables [16], [47], [48].

- Profiling. To keep an eye on performance during development, it is important to *profile* applications.

  Some IDE's, such as Flash Builder and FDT, have advanced profiling functionalities which may help locating performance bottlenecks. Additionally, there are ActionScript3 frameworks that help in profiling applications [46]. Finally profiling can be done by hand using the `trace()` debug statement in combination with `getTimer()` statements and some simple math. However, there are area's in the FlashVM that are almost impossible to profile, except by evaluating the user experience (hiccups, hangs, frame drops etc.).

  Recently, a new performance measurement library was released, named FrameStats [21]. Unfortunately, because it was released only after my internship ended, I didn't have the chance to try it, but it apparently is very powerful, and should definitely be used in future research.

CHAPTER 4

DYNAMIC GRAPHICS

4.1 Introduction

During the first part of my internship, I focused on answering the first research sub question (as listed on page 11). In this chapter, I describe the problems and possibilities related to implementing dynamic graphics in the Mijn naam is Haas game and workflow, in other words, the run time compositing and manipulation of visual representations in the game and the workflow of importing these visual representations into the game.

This research has resulted in a VisualArchitecture for implementing dynamic graphics in the Mijn naam is Haas game and workflow. It supports multipart objects, variants of the same objects, shared mood palette(s) and world layering. In the first sections of this chapter, all terminology and the structuring of the VisualArchitecture is explained. In the concluding sections, the VisualAssetsImporter, a workflow application that can be used to import, organize, and package Visuals, is discussed. For clarity, I will only discuss the final versions of those results in this thesis.

This chapter uses a strict vocabulary to identify the different parts of the VisualArchitecture. All relevant terms have the prefix *Visual*. In situations when the context has clearly been established, and it is clear that Visuals are the subject, the architectural elements below may also be referred to without the *Visual* prefix.

Because the terminology used in the dynamic graphics architecture is also used in the discussion of the design decisions, I first present the final dynamic graphics architecture, before moving on to the design decisions.

4.2 Final dynamic graphics architecture

A ModelObject (or GameObject) is one model object in the Mijn naam is Haas game world, for example Haas, Tree or Bridge. The VisualArchitecture describes and organizes how a ModelObject is represented on the screen. The VisualArchitecture is defined by the following terminology:

- Visual: a prefix relating to anything Visual.
- VisualArchitecture: the VisualArchitecture consists of a few parts: a VisualStructure, VisualLayer(s) and VisualMood(s).
- VisualAsset: one image file containing a VisualFrame. All VisualAssets that belong to a certain ModelObject should have the same size (width, height) in order to allow smart cropping and positioning (storing localWidth, localHeight, localX, localY with a VisualFrame's BitmapData). All VisualAssets should be organized in the VisualAssetsFolder with a certain folder structure to be importable by the VisualAssetsImporter. During the import process, every VisualAsset is transformed into one VisualFrame.
- VisualAssetsFolder: the folder containing all image files that have been exported by the visual artist. For more information about the folder structure, see the section about the VisualAssetsImporter below.
- VisualAssetsImporter: the application used to import VisualAssets from the VisualAssetFolder, organize them, and package them into (a) VisualPackage(s).
- VisualPackage: a (set of) VisualFrame(s) packaged to be used in the game. VisualPackages should be

'smart distributable', which means that two ore more packages can be combined to form one bigger VisualPackage containing the contents of both small packages.

- VisualStructure: the data structure describing how different VisualFrames are related to each other.

  The VisualStructure consists of a hierarchy of VisualCategories, VisualObjects, VisualVariants, VisualParts, VisualPalettes, VisualAlternatives, VisualAnimations, and VisualFrames.

- VisualObject: (or 'visual') a Visual representation of a ModelObject. Every ModelObject is represented by one VisualObject. Every VisualObject has one or more VisualParts. A VisualObject has a number of VisualVariants.

- VisualPart: a part of a VisualObject. For example, in case of a Tree, the following parts might be distinguished: Canopy/Leaves, Stem, Branches. Every VisualPart has one or more VisualAlternatives. Every VisualPart lives on a certain VisualLayer. A VisualPart can have up to one VisualPalette for every global VisualMood.

- VisualMood: A state in which the game world is, that can be reflected in the appearance of individual VisualParts with the help of VisualPalettes.

- VisualPalette: A set of modifier values (brightness, contrast, saturation, hue) that determines how a certain VisualPart is to me modified when the game world is in a certain VisualMood. If no VisualPalette is defined for a certain VisualPart and a certain global VisualMood, the VisualPart's appearance is not modified, and the 'default' images are used.

- VisualLayer: the game world can contain any number of VisualLayers (although it is best to not use too many), indicating the order in which Visuals are to be drawn to the screen.

- VisualAlternative: an alternative for a VisualPart. For every VisualPart, at most one VisualAlternative is selected (the selectedAlternative). VisualAlternatives can be used in any number of VisualVariants of the same VisualObject. Every VisualAlternative contains a number of VisualAnimations.

- VisualVariant: In order to make it possible to combine different VisualAlternatives for the VisualParts of a VisualObject, but not allow all possible combinations, VisualAlternatives can be used in any number of VisualVariants. At any given moment, for all VisualParts of a VisualObject, only VisualAlternatives that can be a part of the VisualObject's currentVariant can be selected as valid currentAlternatives.

- VisualAnimation (replaces Posture(s)): A collection of VisualFrames composing a certain state a VisualAlternative is in, for example 'walking' or 'eating'. VisualAnimations can be looped or non-looped. VisualAnimations that have their *looping* property set to true will be restarted from their first frame when they complete, non-looped VisualAnimations will stop.

- VisualFrame: a frame of a VisualAnimation. Every VisualFrame is loaded from zero or one VisualAsset(s).

4.3 Design decisions

In this final architecture, adding dynamics to the visuals (VisualObjects) is done in multiple ways: using color transformations (VisualMood(s)/VisualPalette(s)), by dividing visuals in multiple parts (VisualParts) and choosing between multiple alternatives for each part (VisualAlternatives), and finally by distributing VisualObjects across multiple layers. Here follow all important design decisions, and their drawbacks and alternative implementations.

- Multiple VisualParts per VisualObject, with multiple alternatives per part, allow for many combinations. Another option is to choose not to include a certain part in some circumstances. In that case, that part is using the 'empty' alternative. This allows for more flexibility in the use of parts and alternatives.

- In the old graphics architecture of the Mijn naam is Haas game, some VisualObjects have *posture(s)*. A posture was a certain state of a visual, for example 'walking' or 'eating'. Most postures had their own animation(s). In the new architecture, postures are replaced by VisualAnimations, which function almost exactly the same as postures did. A problem is, how do postures relate to parts? Should parts have postures/animations, or should postures/animations have parts? The second options seemed to be a bit more logical than the first one, so in the final architecture VisualAnimations are placed below VisualParts in the hierarchy of the VisualStructure.

- But regardless of which option is chosen, the fact that there can be postures as well as parts (which have alternatives) leads to an explosion in the number of VisualAssets required. Therefore, it is in practice better not to use multiple postures and multiple

parts in the same visual. Effectively, the safe choice is: one VisualPart with multiple VisualAnimations; or multiple VisualParts with only one VisualAnimation per part. Theoretically however, the architecture also supports multiple VisualParts with multiple VisualAnimations (postures).

- A goal in future versions of Mijn naam is Haas is to distribute visuals over multiple layers (in the currently released games, visuals can only be placed in front of, or behind the main character Haas). The new VisualArchitecture supports multiple VisualLayers. It is also possible to distribute different parts of a VisualObject over different VisualLayers, so that for example characters can move (walk) before certain parts of a visual but behind other parts of the same visual. This gives more depth to the visual world, while not making the game controls (or more specific the clearness of where a user can draw) more difficult and chaotic. A question that yet has to be answered is how many layers to allow. Theoretically the new VisualArchitecture support any number of layers, but in practice it will probably be best to limit this number of layers to a certain reasonable amount (mainly for performance reasons).

- Color variations can contribute a lot to the visual diversity of the game. In line with the 'intelligent design' principle for Mijn naam is Haas, we however don't want to surrender too much control over the resulting visual style to the game engine itself. The solution for this problem are VisualMoods and VisualPalettes. The idea is that colors should not be changed (pseudo-)randomly, but based on color profiles (VisualPalette(s)) that are predefined. This way, the designer retains control over the visual style. Every VisualObject always has the same VisualMood

applied, but the VisualPalette selected by the global VisualMood is defined for each VisualPart individually.

In other words: a VisualMood can be 'applied' to a VisualObject, which will effectively apply to each VisualFrame of each of the VisualParts of that VisualObject the VisualPalette for that specific VisualMood. Every VisualObject has a 'default' VisualPalette: the VisualFrames as they are by default. A VisualPalette effectively defines a set of color transformations that indicate how to get from the 'default' VisualFrame to the VisualFrame as effected by a certain VisualMood. VisualPalettes are implemented in the final VisualArchitecture as ColorMatrixFilters with four parameters: brightness, contrast, hue and saturation.

Another method to implement VisualPalettes (that was not implemented for performance reasons) is by using a color replace list, containing colors in the 'default' palette and their related colors after the VisualPalette has been applied. To apply a certain VisualPalette, this list is traversed, and every color is replaced according to the list (taking into account a certain precision parameter, so that colors that are 'like' the replaced color are also modified). A color replace list can be constructed by:

1. generating a color palette with distinct colors that occur in the 'default' palette.
2. selecting zero or more colors in this palette.
3. choosing a new color (and a precision value) for each of those selected colors.

The problem with this solution is relatively bad performance, especially when a lot of visuals need to have their VisualPalette (re)applied at the same time.

The advantage of this approach over the implemented ColorMatrixFilters is that while the ColorMatrixFilters perform better, the color replace lists give a more fine grained control over the visual style of the Mijn naam is Haas game.

- The final VisualArchitecture allows any VisualObject to be dynamic. But it is not required for every VisualObject to be dynamic. Which objects could be made more dynamic? Probably the best objects to make more dynamic are objects that are used many times (multiple instances). Variations in these objects will lead to the most effective enrichment of the visual world. These objects are primarily objects that are part of the (natural) world, such as grass, trees and possibly animals.

  In contrast, objects with a clear identity (that are often only used one instance at a time), such as actors (Haas, Sofie, etc.) should look the same every time they appear, and should thus not be dynamic.

  As it turns out, the objects that are the best candidates to be made dynamic are mostly those that don't have postures. This fits nicely with the earlier named 'best practice' to not use VisualAnimations and VisualParts/VisualAlternatives with the same VisualObject.

- Ideally, it should be possible to influence the 'dynamics' of a visual in the Mijn naam is Haas application code, to request a dynamic graphic with certain properties (like 'a high thin tree with only a few leaves and no apples'. It should however also be possible to request just a certain type of visual (like 'a tree'), and have the system determine the specifics by itself. The VisualArchitecture supports this.

- The VisualArchitecture also introduces a number of

workflow related decisions. In the currently released games, visual assets are imported into the Mijn naam is Haas game via .swf library files. While this works well and is quite fast, it is not very elegant and flexible. As an alternative, visual assets can be loaded, serialized, compressed and finally packaged into fast-loading VisualsPackages. Assets (png files) should be organized in the VisualAssetsFolder (root) in a hierarchical way, using this folder structure:

```
root.category.object.part.alternative.animation.frame
```

- Some visuals in the current swf library have Flash frame jump logic in them. This logic is not accounted for in this new system, except for the option to set the looping to *true* or *false*, causing the current animation to loop. Additionally, telling a VisualObject to animate will cause its current Posture or VisualAnimation's parts to be animated. When an animation completes, an Event will be dispatched, allowing the game engine to take appropriate action. Empty frames at a few parts should be possible for better performance.

- In the same manner, some of the visuals currently have audio effects associated with them in the swf library. This audio is not taken into account in the proposed new assets organization, and should be handled by another part of the game engine.

- The organization of the VisualStructure and the storage of the pool of actual BitmapData objects have been designed as two separate software components, the AppearanceProxy and VisualRenderProxy (previously named BitmapDataProxy) respectively. The AppearanceProxy only administers the organization of visual structures, and the VisualRenderProxy (the component that also does the Rendering) stores the

actual BitmapData objects so that they are quickly accessible during the render process. This makes the BitmapData storage very scalable and flexible, but at the same time allows the AppearanceProxy to keep a low footprint (and be loaded fast in its entirety at the start of the game).

The hierarchical relations between the architectural elements are reflected in the following diagram, which also shows their relation to the Model, a connection made by the AppearanceProxy software component. See also the diagram of the VisualAssetsFolder structure on page 48 (figure 4.2).

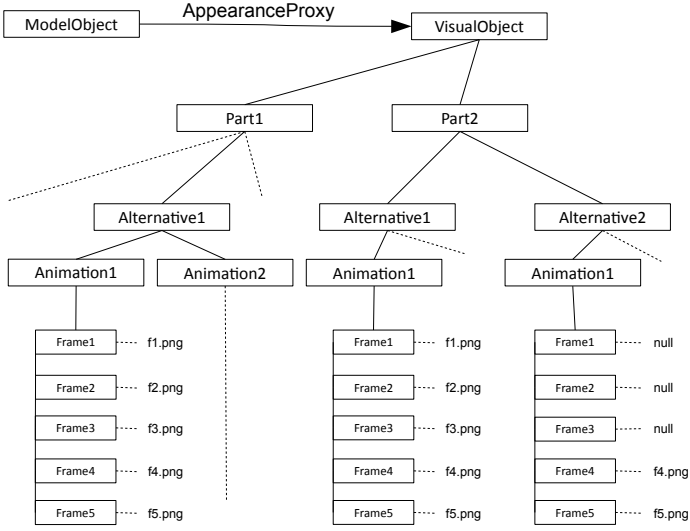

Figure 4.1: From model to visual.

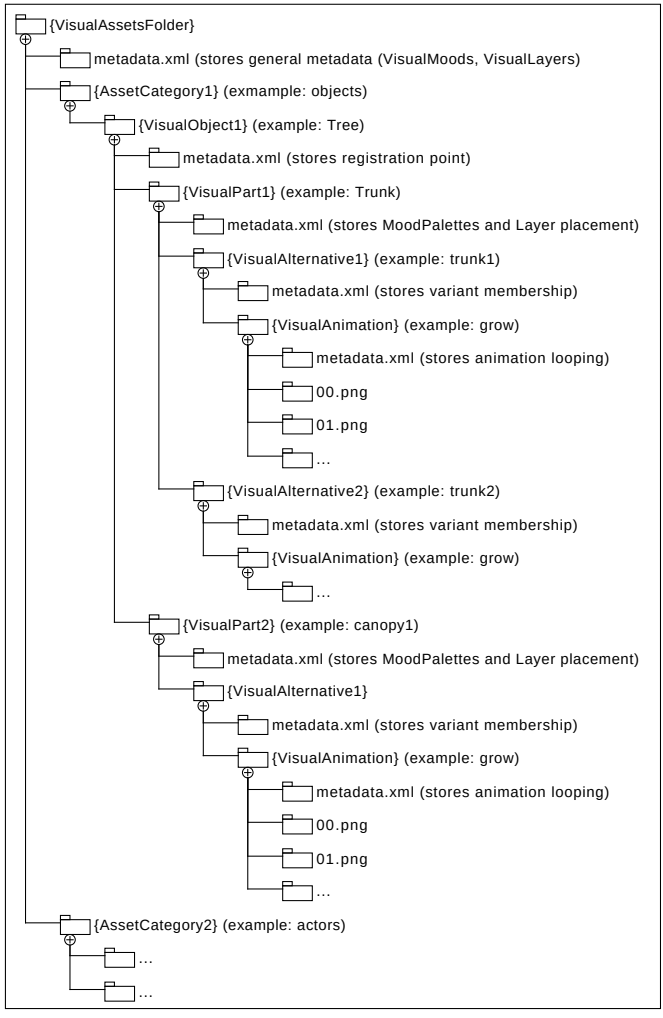

Figure 4.2: The VisualAssetsFolder structure.

4.4 Graphics workflow: importing and rendering

In the process of implementing this VisualArchitecture, it is important to take the complete graphics workflow into consideration, essentially the complete life of a visual from drawing board to game element. There are multiple parts to this workflow. Of course the most important part of implementing dynamic graphics is the rendering of the actual Visuals in the game at runtime. But because good preparation of the Visuals makes the actual rendering process easier and more powerful, I will discuss the complete workflow of getting an image from artist to rendering: the life of a visual. In general, two distinct processes can be distinguished, Importing (A) and Rendering (B):

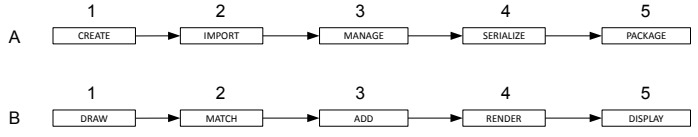

Figure 4.3: The Import and Render pipelines.

This distinction of the two processes leads to two separate implementation tasks:

1. The construction of an application (external to the game) that handles the importing process. This application is henceforth referred to as the *VisualAssetsImporter*.

2. Building component(s) (within the Mijn naam is Haas game) that handle(s) the adding of dynamic graphics and rendering of the game world. I have created multiple software components for the different tasks. The component that handles the management

(adding, removing) of the dynamic graphics is called the *VisualSceneMediator*. The VisualSceneMediator consults the *AppearanceProxy* for information on how to composite visuals and make them dynamic. The component that renders the game world is named the *VisualSceneProxy*.

A: the importing process

The Importing process is performed at development time by the VisualAssetsImporter, and is as such less time and performance critical than the Rendering process. Within the context of my internship, step A2 and A3 are the most important, while step A4 and A5 are side projects for getting the visuals from VisualAssetsImporter into the game.

Step A1: create

After visuals are drawn by the artist and animated by the animator, they are exported as PNG files to an export folder, called the VisualAssetsFolder. The PNG VisualAsset files should be organized in a clear hierarchy (see the image of the folder structure on page 48).

Step A2: import

Next, this VisualAssetsFolder is loaded by the VisualAssetsImporter, which does this by looping over the folder structure and storing all frames into memory. Smart cropping is applied to remove transparent 'whitespace' from the frames, in order to have them use less money. In this process, the localX and localY values of these cropped Visuals are saved. All Visuals belonging to the same VisualObject are stored together.

Step A3: manage

When all Visuals have been loaded, they can be managed in the VisualAssetsImporter application. The following tasks can be performed: (1) set, edit and store each VisualObject's registration point; (2) test *VisualAlternatives* (skip through all alternatives of a VisualPart); (3) edit, test and store VisualPalette's for each VisualPart; (4) play and skip through all animations; (5) edit for every VisualPart the layer on which it is rendered.

Step A4: serialize

All individual images are converted to raw image data (`ByteArray`) and compressed using the lossless *zlib* compression that is built into the FlashVM.

Step A5: package

Finally, the images can be exported to (a) VisualPackage(s) that can be opened quickly by the game, either packaged with the game or from a web API. This packaging is done in a smart way, so multiple VisualPackages can be concatenated to form a combined bigger VisualPackage.

B: the rendering process

The rendering process is 'executed' at run time within the Mijn naam is Haas game by the ObjectMatchProxy, AppearanceProxy, VisualSceneMediator, VisualRenderProxy and other components. Step B1 is handled by the ObjectDrawer component(s) and step B2 by the ObjectMatchProxy. For the purpose of adding dynamic graphics, step B3 is the most important.

Step B1: draw

In this step, the player draws something in the Mijn naam is Haas game world.

Step B2: match

Next, the player's drawing is matched by the ObjectMatchProxy. This results in a ModelObject being added to the game world model.

Step B3: add

This step mainly takes place within the VisualSceneMediator and AppearanceProxy. When the user has drawn an object and it has been matched to a certain type of object, the corresponding VisualObject is selected and an alternative is selected for each VisualPart. After this selection has been done (either on the basis of parameters from the ObjectMatch, or for some alternatives (for example which branches to use) more randomly), the required VisualPackages are fetched if they are not already available.

For performance reasons it is best if all required VisualPackages are already available (preloaded). The new VisualObject is added to the display list. The add step is shown in the following MVC diagram.

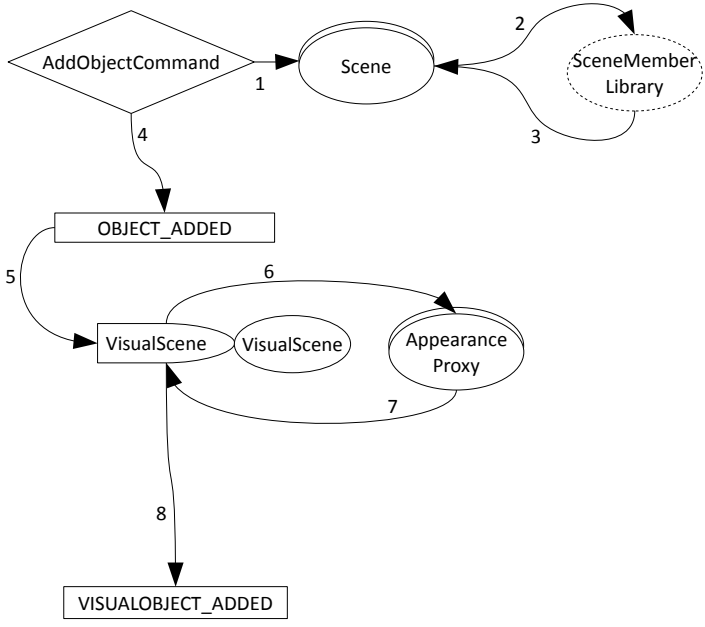

Figure 4.4: The process of adding a VisualObject

When the AddObjectCommand adds a new object, it first (1) adds a ModelObject to the SceneProxy (with the help of the SceneMember libary (2) and (3)). Next, it (4) sends an `OBJECT_ADDED` notification, which is (5) picked up by the VisualSceneMediator. The VisualSceneMediator (6) consults the AppearanceProxy to (7) retrieve a suitable VisualObject for the ModelObject and adds it to the VisualScene. Finally it sends a `VISUALOBJECT_ADDED` notification.

Step B4: render

The render process is managed by the VisualSceneMediator, but takes place mainly within the VisualRenderProxy.

First all VisualFrames in the display list are sorted (based on VisualLayer) and the renderer will loop over the displayList and copy or blit all VisualObjects to the main Bitmap. This is done by copying the currentFrames from all selected VisualAlternatives (one for each VisualPart) to the main BitmapData. In this step actions like z-/layer-sorting, clipping, culling, VisualPalettes, projection and rotation are performed if these are necessary. This step is examined in greater detail in chapter 5.

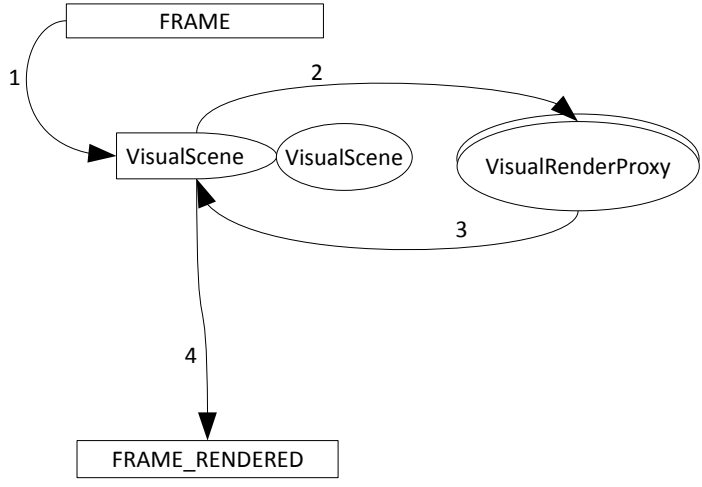

Figure 4.5: The render step of the rendering process

Whenever a frame is to be drawn, a `FRAME` notification is sent by the game core timing mechanisms (TimerProxy). The VisualSceneMediator is (1) interested in this notification and (2) responds by constructing a sceneGraph with the help of the VisualRenderProxy and having the VisualRenderProxy render this sceneGraph. The result (3) is returned back to the VisualSceneMediator which displays it in its viewComponent (VisualScene) and (4) dispatches

a `FRAME_RENDERED` notification.

Step B5: display

The final rendering step consists of instructing the default `flash.display` renderer to show the resulting Bitmap on the game screen.

4.5 VisualAssetsImporter

The VisualAssetsImporter is an Adobe AIR application that can be used to import VisualAssets from the VisualAssetsFolder, manage VisualVariants, manage VisualLayers, distribute VisualParts over VisualLayers, manage VisualMoods, edit VisualPalettes, export VisualPackages, and perform other Visual related tasks. Because all images that are related to the same VisualObject should have the same size, relative positions of the different parts of an object can automatically be detected. The application also enables editing of important metadata such as every VisualObject's registration point in a GUI.

The VisualAssetsImporter performs a 'smart crop' on every frame, stripping away transparent pixels to save memory and bandwidth, while preserving local positions of visuals. The application finally performs 'smart packaging', resulting in VisualPackages that can be concatenated to form bigger packages. These packages can then either be supplied with the game, or fetched from a web service (php or something faster in any requested granularity. Ideally, this back end system is integrated with other back end systems, such as the Intelligent Tutoring System [56].

Importing

When the VisualAssetsImporter is started, it will automatically check the user's `Documents` folder and search

for the `VisualAssetsFolder` (see the folder structure diagram on page 48), and use that folder as its main working directory. It will loop through the folder structure, store the VisualStructure, and convert all VisualAssets to VisualFrames. It will load any `metadata.xml` files it encounters, loading additional metadata for the different Visuals. These metadata files could theoretically be edited manually with a basic text editor, but in principle all metadata can be managed from within the VisualAssetsImporter.

Windows and panels

The VisualAssetsImporter features two windows, the *main window* and the *world preview window*. Each of those windows contains a few different panels.

The main window contains a titlebar with the word 'VisualAssetsImporter' and the location of the VisualAssetsFolder, and three panels. On the left is the visuals panel, which contains selectors for the VisualCategory and VisualObject, a selector to select the current active VisualMood (and buttons to rename or add VisualMoods), a button to start editing VisualLayers, and a button to save all changes to disk (to the VisualAssetsFolder). Finally the visuals panel contains a button to export the Visuals to a VisualsPackage. The parts panel contains controls to edit all VisualParts of a VisualObject independently, and a button to add copies of the edited VisualObject to the world preview. The palettes panel contains controls to edit a VisualPalette. The main window cannot be resized.

The world preview window contains a world preview panel that takes up most of the window, and a display list panel, which contains controls to manage the VisualObjects that are in the world preview at any given time. The world preview window can be resized to occupy a whole screen.

Usage

When an object is selected using the selector in the visuals panel, that object will become the CurrentObject, the object that is currently being edited. The CurrentObject is automatically added to the world preview, and can't be removed from it. Instead, it is automatically replaced whenever another object becomes the CurrentObject. Persistent copies of the CurrentVisual can be added to the world preview by clicking the 'Add Visual to Preview' button in the parts panel.

Additionally, when a VisualObject becomes the CurrentObject, controls to configure its VisualParts, VisualAlternatives and VisualAnimation are added to the parts panel. For each part, a layer can be selected onto which that part will be placed in the world, and one of the alternatives can be selected as the selected alternative (selectedAlternative) for that part. By clicking the 'Variants' button, a selection of the VisualVariants this VisualAlternative is a valid alternative for can be made. For the selectedAlternative, the active VisualAnimation (Posture) can be changed. The checkbox next tot the posture selector can be used to set the animation behavior of the posture to loop (restart from the first frame after the VisualAnimation completes) or not looping (stop after one animation cycle). For each part, the VisualPalette corresponding to the globally selected VisualMood can be edited by clicking the Edit Palette(s) button.

Using the 'Package' button in the visuals panel, all Visual information (the VisualStructure, the VisualMoods, the VisualLayers and the BitmapData (VisualFrames)) can be exported for use in the Mijn naam is Haas game. This functionality has been implemented, but has not optimized for smart packaging, because that is outside the scope of my internship.

4.6 Results: architecture and workflow

This part of my internship has resulted in a powerful dynamic graphics architecture and a very useful visual assets workflow application. The VisualArchitecture provides a lot of possibilities for dynamic graphics, and all required functionality of the VisualAssetsImporter has been implemented. Of course, the VisualArchitecture could also be partly used, as there are no dependancies between the different parts of the architecture or between the different methods of dynamic graphics the architecture offers. Similarly, the VisualAssetsImporter can easily be improved or changed as it is built using the PureMVC framework.

Use case diagram

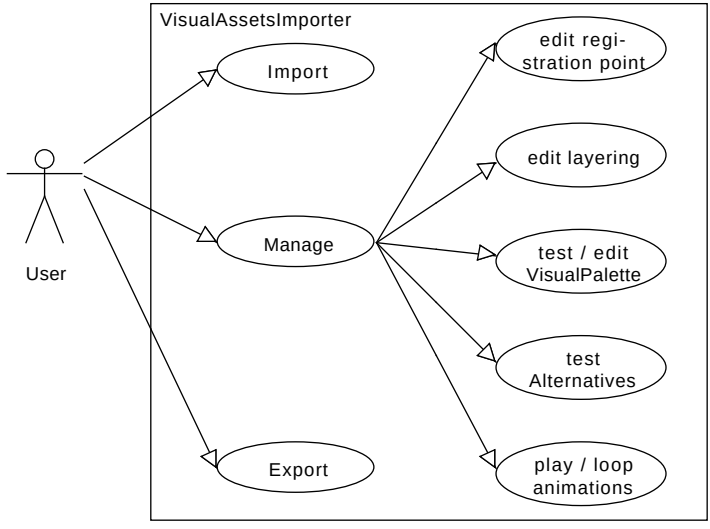

Figure 4.6: The VisualAssetsImporter use case diagram

Development wireframe

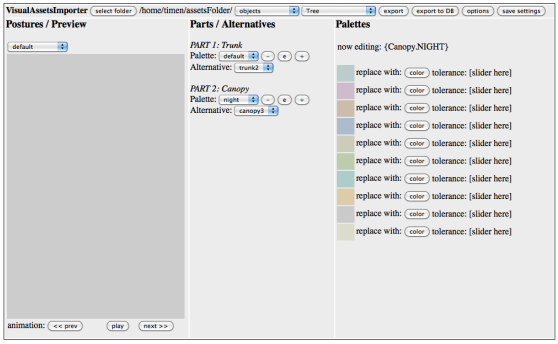

Figure 4.7: The VisualAssetsImporter wireframe

Screenshots

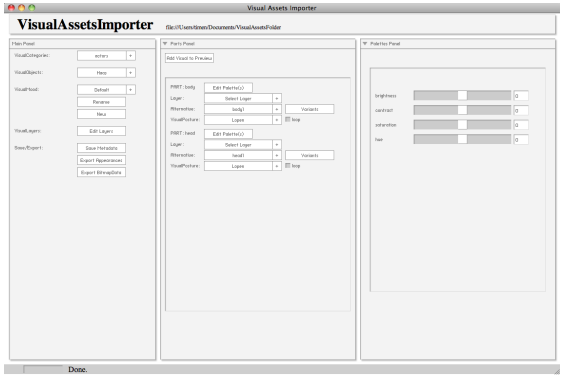

Figure 4.8: The VisualAssetsImporter's main window

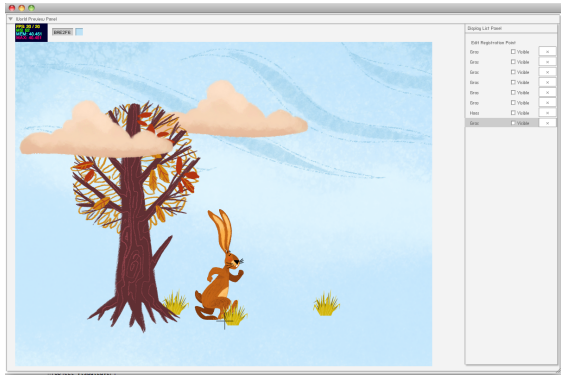

Figure 4.9: The VisualAssetsImporter's review window

Bugs, additions and improvements

There is a number of features and improvements that could be implemented that would make the VisualAssetsImporter a better program. For possible future use I list them here in a very short manner:

- Import empty frames based on filename (0.png, 1.png, 5.png should create 3 empty frames), could be implemented in VisualAnimation or VisualFrame.
- VariantsList per VisualObject + variant checkboxes per VisualPart.
- VisualObject filtering so that only VisualAlternatives that can be an alternative for the selected VisualVariant are shown.
- More and better visual feedback on which elements (Parts, Objects, Palettes, etc.) are currently being edited.

- Animation controls per posture, and/or global animation controls.
- Animation for all visuals in the preview window, instead of only for the the CurrentObject.
- Support for empty VisualAnimations, default VisualAnimations and empty frames.
- Implement a better world preview renderer.
- A button to delete VisualMoods.
- Check consistent part/alternative ordering.
- Multiple PaletteAlternatives per VisualPalette.
- Undo functionality.
- Move preview objects and registration point using arrow keys.
- Palettes copyable between parts.
- Multiple LayerAlternatives / LayerTypes

CHAPTER

5

RENDER ENGINE

5.1 Introduction

During the second part of my internship, I focused on answering the second and third research sub questions (as listed on page 11). After the structural design of the new dynamic graphics architecture for the Mijn naam is Haas game was completed (including an application for editing, testing and packaging visuals), the goal was design and build (a part of) a render engine that can render these dynamic graphics in the fastest way possible. This chapter discusses the render engine architecture and implementation I have developed during my internship, based on the research listed in chapter 3.

The wish for a new render engine for the Mijn naam is Haas games did not just originate from the requirement of adding dynamic graphics to the game. Other reasons to investigate the possibilities for a new render engine are the fact that a custom engine would allow for more game specific tweaking, could be optimized for use in the web service version of the Mijn naam is Haas game, could improve the overall graphical assets workflow, and make the game world bigger and more free. While researching, designing and implementing the new render engine, I took into account all those requirements, but the implementation of dynamic graphics remained the core of the project.

5.2 General render engine theory

Usually, rendering is an expensive task in terms of the machine power required. Therefore, when it comes to render engines, performance and functionality are eternally battling for importance. A lot of methods, techniques and tricks are used in creating render engines, in order to make them more powerful as well as faster. Here, I discuss some of these.

parallax parallax is a technique used to 'fake' 3d depth in 2d worlds. It works by rendering multiple layers in front of each other. The key point is that whenever the camera moves, 'deeper' layers move slower than layers closer to the camera. This suggests depth.

animation animation is the simulation of internal movement within elements of the visual world. It works by showing multiple still images that slightly differ quickly after each other. This suggests change within the visual element.

blitting blitting is the strategy of translating manipulations of visual elements into manipulations of blocks of bits. Usually, blocks of bits that represent a visual object are copied to one big block of bits representing the world. After this has been done for all objects in the game world, the big block of bits has become a visual representation of the game world. This method works very well with all kinds of caching strategies, as the small source blocks can easily be stored and reused.

caching caching is the storage and reuse of elements during the render process. It works by storing the result of each task. This way, tasks that have not changed since the last iteration don´t have to be performed again. Instead of performing a task again, the result that was obtained last time can be reused. Improved caching strategies try to predict which tasks are likely to be reused, and store just the results of those tasks.

scrolling scrolling is the rendering of a large part of the world, and then showing just the visible part of this large part. When the camera position changes, the world doesn't need to be rendered again. Instead, another part has to be shown.

dirty areas using dirty areas (also known as redrawing areas), the render algorithm can be improved so that only parts of the game world that have changed need to be re-rendered. This saves a lot of rendering time if there are only a few or small local changes, but is a lot of work if there are many changes in the global game world.

culling / visibility testing culling is the process of selecting a subset of render tasks from a larger set of possible tasks. This can be used to select only relevant tasks, and discard tasks that have no influence on the end result of the rendering process before actually completing them. This can save a lot of time. A widely used example of a culling strategy is visibility testing, which checks (using fast mathematical algorithms that take into account position, size, viewPort and other properties) for each visual object that is to be rendered whether or not it would be visible in the end result. The render engine can than decide to drop all invisible render jobs from the render queue, saving a lot of execution time.

garbage collection garbage collection is the 'automatic' removal of unused data structures from memory, creating more room for new tasks. Garbage collection can be very important if caching strategies store a lot of previously generated results. Results that are not likely to be used again have to be cleaned up.

display list a display list is a data structure storing all (visual) objects in a given world or scene. This list is used as administration of the world state, and is used as a starting point for rendering jobs during the rendering process. A scene graph is a display list with a more complex structure, that for example also stores relations between objects and sub objects.

viewport a viewport is the area on the users screen where the resulting visual representation of the world is shown to the user. The viewport is used (together with the view rectangle, which stores the part of the world which is visible) to determine which objects need to be rendered, and which are visible on screen.

FlashVM render engine the FlashVM render engine is the default Flash render engine. It allows programmers to add various types of DisplayObjects to a display list. This display list is then rendered automatically by the Flash virtual machine. The programmer just has to set certain properties such as `x`, `y`, `rotation`, `aplha`, etc. and the FlasVM render engine takes care of the actual processing of these values. It is not very much customizable, and performance varies per operating system platform.

5.3 A new Mijn naam is Haas render engine

This section describes the core components and architectural elements of the new Mijn naam is Haas render engine. It also lists the basic workings of those components and elements.

- The new render engine is built around the dynamic graphics VisualArchitecture described in chapter 4.
- VisualScene: A scene in the game (or: all objects in the game world at a certain moment.)
- AppearanceProxy: The proxy component responsible for managing the VisualStructure, VisualLayers and VisualMoods.

  - On game startup, the VisualStructure, VisualLayer(s) and VisualMoods are loaded into the AppearanceProxy. They will be serialized from the VisualAssetsImporter, and deserialized on game startup.

- BitmapDataProxy: The proxy component responsible for managing the BitmapData (VisualFrames).

  - On game startup, the visuals that are expected to be needed in the game session are loaded from one or more VisualPackage(s). The package importing functionality has been implemented, but it has not been optimized for 'smart packaging' yet. Possible smart packaging technologies include static package files (serializing, compessing and packaging BitmapData during development), SQL tables and databases, http repository and/or dynamic server side packaging (constructing client-specific packages on the fly from tables or data files).

- VisualSceneMediator: The mediator component responsible for managing changes in the sceneGraph, and setting the renderer to work on FRAME notifications.

  - When a MODELOBJECT_ADDED notification is sent, the VisualSceneMediator will respond by asking the AppearanceProxy for the appropriate VisualObject that is to be used to represent the ModelObject in the VisualScene. The VisualSceneMediator than adds the VisualObject to the VisualRenderProxy's sceneGraph.
  - When a MODELOBJECT_CHANGED notification is sent, there will have to be some kind of way of updating VisualObjects when their cor-

responding ModelObject changes (for example position).
  - When a MODELOBJECT_REMOVED notification is sent, the VisualSceneMediator will respond by removing the related VisualObject from the VisualSceneProxy's sceneGraph.
  - When a FRAME notification is sent, the VisualSceneMediator will respond by instructing the VisualSceneProxy to render its sceneGraph to the viewPort.

- VisualSceneProxy: The proxy component responsible for rendering the visual scene.
  - The VisualSceneProxy has a *sceneGraph* (a (hierarchical) list of all VisualObjects in the VisualScene), *viewPort* (a Bitmap instance) and a *cameraRect* (a certain region of the world that is to be rendered, and from which a zoom value can be calculated).

- The VisualSceneProxy can be implemented using different technologies (as examined in chapter 3):
  - FlashVM: the FlashVM DisplayObject renderer.
  - ActionScript3: use a custom ActionScript3 renderer using `draw()` and `copyPixels()`;
  - Alchemy: use an external C library to do the actual rendering. This library is compiled to a .swc and used by the VisualSceneProxy. To speed up this rendering, the BitmapDataProxy will also have to be implemented in Alchemy/C.
  - HaXe: use an external HaXe library to do the actual rendering. This library is compiled to a .swc and used by the VisualSceneProxy. To speed up this rendering, the BitmapDataProxy will probably also have to be implemented in HaXe.

5.4 Implementation

The VisualSceneProxy can theoretically support up to four render modes, each using a different technology. The selection of the different render modes is listed here in pseudocode:

```
if (renderMode=="Alchemy") { //use Alchemy
  //loop over sceneGraph
  //check visibility (incl. rotation size change)
  //create multidimensional queue
    //(DrawFrame(screen position, rotation,
    //      palette(4values), bmdPointer))
  //concatenate to 1 queue
  //copy queue to alchemy Memory
  //render queue
    //lib.render()) does position, rotation,
    //      palette(4values) (unzip?)
  //copy result to screen

} else if (renderMode=="HaXe") { //use HaXe
  //loop over sceneGraph
  //check visibility (incl. rotation size change)
  //create multidimensional queue
  //render queue, per item:
    //get bmd from bmdproxy (or cache pointer)
    //palette (optionally: cache result)
    //rotate

} else if (renderMode=="FlashVM") { //use FlashVM
  //add all VisualFrames as DisplayObjects
  //        to the VisualScene
  //the VisualArchitecture isn't really optimized
  //        for this type of rendering

} else { //renderMode =="as3" //use ActionScript3
  //use naive BitmapData.draw() and copyPixels() rendering
  //loop over sceneGraph
  //check visibility (incl. rotation size change)
    //(DrawFrame(screen position, rotation,
    //      palette(4values), bmdPointer))
  //concatenate to 1 queue
  //copy queue, render everything using copyPixels
}
```

Here follows a breakdown of the workings of the new render engine game components as implemented within the Mijn naam is Haas game. Part of the implementation centers around loading assets when the game initializes, a second part is related to the run time adding of visuals to the world based on input from the user input (as matched by the ObjectMatch), and a third part (consisting of the TICK and FRAME steps) deals with the actual drawing of a frame of the visual appearance of the world on the screen.

Game Initialization

- Using the *VisualAssetsImporter*, the visual structure and bitmap data can be organized and exported to two different files: an Appearances file and a BitmapData package file. The Appearances file contains the complete VisualArchitecture structure. The BitmapData package file contains all BitmapData objects serialized to a ByteArray and compressed using zlib.

- In the *LoadAssetsCommand*, the VisualRenderProxy is instructed to load the BitmapData package file.

- Also in the *LoadAssetsCommand*, the AppearanceProxy is instructed to load the Appearances file.

- Both these files are binary files that are imported to their respective proxy's. The BitmapData package file is parsed and the individual ByteArrays are stored in the *bmdStorage* Array in the VisualRenderProxy indexed by their bitmap-data ids The ByteArrays are wrapped into *RenderData* objects that also contain some metadata.

Adding VisualObjects

- Whenever a model object is added by the *AddObjectCommand*, the object is no longer added to the *SceneMediator*, but instead the new *VisualSceneMediator* listens itself to changed (additions and removals) of model objects.
- When the *VisualSceneMediator* detects an addition of an object, it creates a new *VisualObject* (that contains a reference back to the corresponding *WorldMember* model object), and adds this *VisualObject* to the VisualSceneProxy.
- To determine the type and settings of the new *VisualObject*, the *VisualSceneMediator* consults the *AppearanceProxy*, which in turn uses the stored VisualArchitecture structure to decide.
- A similar process is performed to remove objects

TICK notification

- On a TICK notification, playing animations are advanced one frame and completed / looping animations are handled.

FRAME notification

- – Construct SceneGraph –
  - Whenever a new FRAME is to be drawn, the *VisualRenderProxy* is instructed to build a SceneGraph. To construct this SceneGraph, the list of VisualObjects that are currently in the ViewPort (=ViewRectangle) is examined. Since a VisualObject may consist of multiple parts, which may each have a selectedAlternative with a VisualPosture, the bmdId of the currentFrame of

each of those VisualPostures is looked up in the *bmdStorage* Array, which results in a reference to a RenderData object for each VisualFrame.

  - A check is performed to determine if that specific RenderData's BitmapData has been uncompressed already. If not, it will be uncompressed.
  - A *RenderJob* is constructed for each frame that is within the ViewPort. This selection is made with the help of a *isVisible()* method, that checks whether the VisualFrame's visibility rectangle (based on the frame's location and size) intersects with the ViewPort.
  - If a *RenderJob* is within the *ViewPort*, the *RenderJob* is added to the SceneGraph. Otherwise, the *RenderJob* is discarded.

- – Create renderQueue –
  - All RenderJobs are serialized into one large RenderQueue ByteArray. This process could possibly be sped up by using fast memory for the RenderQueue ByteArray.

- – Render –
  - The RenderQueue is read byte by byte and the correct BitmapData is blitted to the main BitmapData screen. This is done using *copyPixels* if no scaling or rotation is required, or using *draw()* if scaling or rotation is required.
  - During this process, the position of the VisualObject in the world, the VisualFrame cropping area, the VisualObject's registration point, the world zooming factor and the ViewPort size are taken into account.
  - An optional VisualPalette is also applied during this step.

5.5 Results: functionality and performance

As planned during the first weeks of my internship, I have implemented (a first version of) a new Mijn naam is Haas render engine, that can help in determining the options for a final new render engine, and functions as a first reference prototype for future versions.

The resulting render engine was written entirely in ActionScript3. The experimental technologies I researched, Pixel Bender, Alchemy and HaXe, didn't perform as I had hoped. Even with small example render tests and implementations, frame rates actually dropped, and the work flow complexity drastically increased. Therefore, I decided to go for a pure ActionScript 3 implementation, which not only fits better within the existing project work flow, but also leaves more room for easy fine tuning. Also, a pure ActionScript3 implementation will benefit more directly from any improvements Adobe makes to the FlashVM implementation.

I would recommend Adobe Pixel Bender only for per-pixel graphics filters, and for large computational tasks that can be 'translated' into many small pixel manipulations. Hopefully Adobe will update Pixel Bender to actually use the video card hardware acceleration in Flash.

Similarly, Adobe Alchemy can be very useful for large calculations, and for using existing C libraries in Flash, but as a replacement of ActionScript3 it doesn't help performance in render engines.

The power of HaXe lies in it's cross platform deployment and optimized compiler, but the result is only faster when very specific features, like the fast Memory, are used. To use these features in the context of render engines, requires to much work flow adaption and creates to much overhead to be feasible in practice.

What all this shows is the need for Adobe to do some-

thing to the performance of the FlashVM, and especially the graphics implementation. They announced that they are working on a better 3D engine, but true hardware acceleration and compiler optimization may actually be areas where more performance improvements can be earned.

The resulting new render engine for the Mijn naam is Haas games supports a lot of features. Like any render engine it supports basic rendering features like positioning, scaling/zooming and rotation. Of those, rotation and scaling are quite expensive operations, as the ActionScript3 function that perform these tasks, `draw()`, is a lot slower than `copyPixels()`, a similar function that doesn't do rotation and scaling. Caching rotation and scaling operations therefore would be a very good strategy. Unfortunately I had no time to implement this during my internship. What makes rotation more complex is the fact that a lot of per frame values, such as relative location of parts (and cropping values) have to be taken into account.

Of course the new render engine supports the three methods of dynamic graphics described in chapter 2: it supports the compositing of VisualObjects using VisualParts, VisualAlternatives and VisualFrames. It supports animation of those graphics. It also supports dynamic colors with the help of per part VisualPalettes and global VisualMoods. This works quite fast, but there is room for improvement in the implementation. Finally, the render engine supports multiple world layers, which basically translates to a depth sorting algorithm in the render engine. This sorting can be further optimized to obtain some performance improvements.

To increase performance, I implemented culling in the form of a visibility test. This test checks for each frame in the display list whether or not it would possibly be visible in the final render result. It does this by calculating a visibility rectangle for every frame and checking whether or not this rectangle intersects with the rendered part of

the world (see figure 5.5). I also implemented a blitting algorithm that reuses the same source bitmap data for each visual of the same type. This means for example that all trees of the same type share one source bitmap data object for their frame. This saves memory.

Because the new render engine was developed in close relation to the complete visual assets work flow process, it deals with deserializing and uncompressing visual assets. It also uses preloading of all visual assets so they load faster when they are needed during the render step.

The new render engine has been implemented next to the existing one, and all graphics that are drawn by the player during free drawing game phases have been transferred to the new render engine. The new render engine was then overlaid onto the old one. This was done to make development easier, because it allowed the piece by piece transition to the new render engine, and a lot of visual and debug feedback during development. It also meant that there was a working game build during every stage of the development process. The render engine has been implemented using the PureMVC framework, just like other game components. However, at some points in the code, code readability has partly been sacrificed to improve performance. In general there is a trade off between code readability (more overhead) and execution speed (less overhead). At a few points in the render engine, the code could be made even faster by sacrificing even more code readability. However, this will require some architectural changes, and possibly also a deviation from the PureMVC framework.

It turned out to be practically impossible to obtain coherent, comparable performance tests of the new render engine. Of course it is possible to perform perform basic tests on small parts of the render process or on data structure, and determine which data type is faster than which when performing a lot of small mathematical calculations

on them. However, rendering is very much unlike small mathematical calculations. Rendering is performing a list of fifty to a few hundred small to medium sized operations, and rendering is not just performing computational calculations. The performance of the render engine greatly varies depending on the context: the number of visuals, the movement, zooming, animation, the user's actions etc. Individual parts of the render pipeline can be tested, but those tests are difficult to compare and do not necessary help in determining the performance of the render engine in practice. In the end, the best measurement is simply the experience of the responsiveness of the game in actual use.

The default frame rate of the Mijn naam is Haas games is 20fps. As long as no rotation or scaling is involved, the new render engine performs really well and easily renders a large number of complex visual objects (multiple parts) with color filters applied that are distributed over multiple layers. However, as soon as rotation and scaling is needed, performance drops significantly. This shows how much an ActionScript3 renderer is still dependent on Adobes choices on the implementation of the `flash.display` renderer. While all independent parts of the render pipeline are quite fast, except for scaling and rotating, the overall render process is, with all functionality enabled, too slow. As long as Adobe doesn't update its compiler and/or improves the implementations of the API calls, the best solution is to focus on part of the functionality and cancel the rest of the functionality. For example, the compositing could be limited to a certain number of parts or a smaller hierarchy, as the amount of work that needs to be done augments with every added part.

But in practice, there isn't really one part of the render process that is the main culprit of the so-so performance. It is really all parts together that perform less than I hoped for. The dynamic color replacements are

relatively fast, because they are based on internal filters (ColorMatrixFilter). The compositing is not very slow as it mainly uses more memory to store the different alternatives and their relations. The retrieval of the selected alternatives is quite fast. The sorting of the visual elements based on their layer is not that difficult either and could be sped up if better native sorting algorithms would be available. The fact that all visual data is preloaded removes all network and filesystem overhead for the cost of extra memory usage and a longer (pre)loading time. Compression and serializing save memory space on data that isn't actually used in the game. The visibility test is complex yet mainly consists of (relatively fast) mathematical calculations. The blitting is powerful and can easily be combined with caching strategies. During my internship I only implemented global caching (by integrating the BitmapDataProxy component into the render engine component) for the source visuals in the blitting algorithm. There is certainly room for additional caching strategies to increase performance further. However, the zooming that occurs in the Mijn naam is Haas game is a problem, as it influences caching strategies a lot. Whenever a zoom-in or zoom-out occurs, all visual elements will have a new scale, and need to be re-rendered at the same time. In other words, the whole cache will be invalidated, causing a lot of work for the renderer at once. Other possible caching strategies include for example the caching of the coloring, scaling, visibility check and animation frames.

Figure 5.1: Screenshot of new render engine in action (1), the new render engine is overlaid on the old engine

Figure 5.2: Screenshot of new render engine in action (2)

Figure 5.3: Screenshot of new render engine in action (3)

Figure 5.4: Screenshot of new render engine in action (4)

Figure 5.5: Screenshot showing the visibility rectangles for some of the visuals

CHAPTER 6

CONCLUSION

6.1 Summary

Serious games are an important part of the multimedia discipline. By mixing education and entertainment, they provide a combination of learning and playing experience. The goal of my internship was to work on the visual appearance of the game world of the Mijn naam is Haas serious game, in order to make it more diverse, dynamic and free.

A common problem in the creation of virtual and digital worlds is a lack of diversity in appearances. For example, when you look at a real forest, no tree is exactly alike, but in virtual worlds, multiple trees often are very much or completely alike, because they are rendered from the same source image as drawn by an illustrator or designer. Another approach at creating virtual worlds is procedural world (content) generation. In this case de appearance of the world is completely generated by mathematical routines. Generated virtual worlds often look too dynamic, and contain too much variation. A virtual world with not enough variation looks 'closed', 'boring' and 'strict', and a virtual world with too much variation is experienced as 'random', 'dazzling' and 'cluttered'. To overcome these extremes, I used 'intelligently designed' dynamic graphics for the Mijn naam is Haas game, which means that the visuals are designed by an illustrator and designer. These graphics are then dynamically altered when they are used in the game world. The goal of this process is to make the game world 'open', 'interesting' and 'organic'.

The internship produced multiple results. First I developed a dynamic graphics architecture that supports dynamic adaption of graphics with the help of shape variations (multiple versions of the same tree with a different shape), color variations (subtle and situation based color differences between the same graphics) and composition variations (multiple layers of depth within the game

world). To support the implementation of this dynamic graphics architecture I developed a work flow application that can be used to import, manage and package visuals and their variations. Next, I worked on the implementation of these dynamic graphics within a new render engine software component for the Mijn naam is Haas game. The Mijn naam is Haas game is being developed in Flash, the de facto internet standard for interactive applications. After researching several additional technologies (Adobe Pixel Bender, Adobe Alchemy, HaXe) that could be used to implement the render engine, I decided to implement the render engine in pure Flash ActionScript. The resulting render engine prototype supports all dynamic graphics functionality of the dynamic graphics architecture. I also spent a lot of time improving the performance of the render engine: I implemented a visibility testing culling strategy and a bitmapdata blitting algorithm with shared source images.

The render engine prototype works very well, but the integration within the Mijn naam is Haas game has not reached the state where it can be used in release build of the Mijn naam is Haas games. Also, a lot of future research can be done to improve the performance further, because the more visual elements are to be rendered, the slower the game becomes. The first prototype provides a lot of practical information about how dynamic graphics could be used within the Mijn naam is Haas game world, and which implementation possibilities work better than others.

6.2 Technical conclusions

The experimental approach of this project has allowed me to investigate a lot of different technologies and possibilities. It is important to summarize the conclusions of all this theoretical work in a few key points. This makes the

general advantages and disadvantages clear.

Returning to the original research questions, we can conclude that there are multiple effective strategies for making graphics dynamic. The three implemented methods, color variations, shape variations and depth variation (layers) all are working very well in practice, and the individual strategies are all implementable with reasonable performance. Using all strategies at the same time however quickly demonstrates the limits to the number of visuals that can be rendered at the same time with playable performance. Because parts of the VisualArchitecture are not dependent on each other, the architecture can be implemented selectively to obtain better results.

While experimental technologies like Adobe Alchemy and HaXe appear potentially very powerful, their use in the field of speeding up render engines is problematic, because these technologies are most effective to improve performance of large sets of repetitive mathematical calculations or memory intensive recurring algorithms, but they do not significantly speed up pixel manipulations and transformations. To bypass the limits of the FlashVM render engine, a pure ActionScript3 blitting render engine that has more specific functionality can be built, but that still not overcomes the relatively bad optimization of the flash compiler.

In the Mijn naam is Haas game, dynamic graphics can really contribute to a more vivid and organic world. In order to obtain the best performance, choices should be made about the most important methods of dynamic graphics, that improve the visual style most.

Personally, I would definitely introduce small (maybe even pseudo random) color differences between visuals that are used very often (such as trees and other natural elements), and also changing the colors using VisualPalettes based on the 'mood' of the game is a powerful yet easy process.

Layering world objects can be a real improvement to the game world, but it also introduces sorting overhead. This overhead can be justified more easy if the effect of the layering is greater, for example if a small amount of parallax would be introduced.

Compositing multi part visual objects introduces most overhead into the graphics work flow and render engine. An alternative to run time compositing could be the pre-rendering different variations of the same object. This will however introduce a lot of extra memory usage and reduce the influence the designer has on coloring individual parts at runtime.

A final improvement that could be made to the Mijn naam is Haas game core to better accommodate dynamic graphics is redesigning the ModelObject architecture and integrating it with the VisualObject architecture, so that one single GameObject architecture is formed. Doing this will remove a lot of updating and organizing overhead.

6.3 Personal internship evaluation

The five months of my internship at Mijn naam is Haas were over before I knew it. My internship was a great experience from beginning to end. From the start I was welcomed with great warmth, and I really liked working in the enjoyable team of professionals. The original planning of my internship was successfully completed a few days before my internship period ended. Of course every completed challenge opens new challenging paths.

Looking back, I am very satisfied with the overall process and results of the internship. While the render engine will not be used in the Haas game in its current stage of development, the work I did during my internship provides a clear overview of the possibilities and impossibilities of dynamic graphics, and functions as a stable basis for future work on the subject.

Before I started my internship I formulated the goal of “proactively working on an innovative product within a dynamic and creative professional environment”. This goal was clearly accomplished. Working in the vibrant and stimulating atmosphere of the Mijn naam is Haas team was great. Besides working in a multidisciplinary team, I also learned a lot about the creative business, developing a large complex game application and of course dynamic graphics and render engines.

For me this internship was the perfect combination of interesting technical research and challenging practical work. Working with experimental technologies and low level render engine implementations was very informative and translating technological possibilities into practical solutions was challenging. Working within a team of great colleagues made the whole internship a very pleasant experience.

6.4 Internship evaluation by Berend Weij

[nog aan te vullen]

BIBLIOGRAPHY

[1] ActionSnippet. *setVector vs copyPixels*. http://actionsnippet.com/?p=1767. July 2009.

[2] Nick Avgerinos. *Understanding Adobe Flash Platform technologies for building games*. http://www.adobe.com/devnet/flashplatform/articles/gaming_technologies.html. Feb. 2010.

[3] Nicolas Barradeau. *Color depth change*. http://en.nicoptere.net/?p=8. July 2008.

[4] Nicolas Barradeau. *Pixel Bender 1 color tweaking*. http://en.nicoptere.net/?p=267. July 2009.

[5] Nick Bilyk. *Optimizing Actionscript and your approach to optimizing*. http://www.nbilyk.com/optimizing-actionscript-3. July 2008.

[6] Corey von Birnbaum. *AS3 to Pixel Bender guide*. http://coldconstructs.com/2009/10/pixel-bender-gap-guide/. Oct. 2009.

[7] Francis Bourre. *full double-buffering alchemist*. http://blog.tweenpix.net/2008/12/04/full-double-buffering-alchemist/. Dec. 2008.

[8] Francis Bourre. *smooth plasma experiment*. http://blog.tweenpix.net/2008/12/16/smooth-plasma-experiment/. Dec. 2008.

[9] Trevor Boyle. *Dynamically changing the palette of an image using Flash (AS3)*. http://blog.webdeely.com/2009/03/storing-custom-strongly-typed-amf-files-for-air-apps/. Oct. 2009.

[10] Lee Brimelow. *Loading Pixel Bender Filters in Flash 10*. http://theflashblog.com/?p=386. May 2008.

[11] Matthew Butt. *Pixel Bender Tutorial*. http://blog.preinvent.com/node/24. Nov. 2009.

[12] Nicolas Cannasse. *Adobe Alchemy*. http://ncannasse.fr/blog/adobe_alchemy. Nov. 2008.

[13] Nicolas Cannasse. *HaXe: Virtual Memory API*. http://www.ncannasse.fr/blog/virtual_memory_api. Nov. 2008.

[14] Chris Deely. *Storing Custom AMF files for AIR Apps*. http://blog.webdeely.com/2009/03/storing-custom-strongly-typed-amf-files-for-air-apps/. Mar. 2009.

[15] Adobe Devnet. *Pixel Bender Technology Center*. http://www.adobe.com/devnet/pixelbender/.

[16] Joa Ebert. *ActionScript optimizations wiki*. http://wiki.joa-ebert.com/index.php/Main_Page.

[17] Joa Ebert. *Apparat: A framework to optimize ABC, SWC and SWF files*. http://code.google.com/p/apparat/.

[18] Joa Ebert. *First results of TAAS*. http://blog.joa-ebert.com/2009/09/01/first-results-of-taas/. Sept. 2009.

[19] Joa Ebert. *Imageprocessing Library*. http://blog.joa-ebert.com/imageprocessing-library/.

[20] A. Eliens and Zs. Ruttkay. "Record, Replay & Reflect : a framework for serious gameplay". In: *Proceedings EUROMEDIA 2009*. Brugge (Belgium), 2009. URL: http://www.cs.vu.nl/~eliens/research/projects/media/paper-replay.pdf.

[21] Elad Elrom. *FrameStats performance monitor tool*. http://elromdesign.com/blog/2010/07/20/performance-monitor-tool-to-watch-internal-of-the-flash-player-during-runtime/.

[22] Elad Elrom. *Using Pixel Bender to calculate information*. http://www.flashmagazine.com/tutorials/detail/using_pixel_bender_to_calculate_information/. Aug. 2009.

[23] Renaun Erickson. *Rendering game assets in ActionScript using blitting techniques and Flash Builder 4*. http://www.adobe.com/devnet/flex/articles/actionscript_blitting.html. Feb. 2010.

[24] Jeff Fulton. *Actionscript 3: Tutorial - BitmapData rotation with a matrix*. http://www.8bitrocket.com/newsdisplay.aspx?newspage=6765. Oct. 2007.

[25] Jeff Fulton. *Flash AS3 Speed Tests: Rendering and Update Models*. http://www.8bitrocket.com/newsdisplay.aspx?newspage=7496. Dec. 2007.

[26] Jeff Fulton. *Tutorial : Creating an Optimized AS3 Game Timer Loop*. http://www.8bitrocket.com/newsdisplay.aspx?newspage=10248. Apr. 2008.

[27] Jeff Fulton. *Tutorial: AS3 Basic Blitting #2 : Rotation - Part 1*. http://www.8bitrocket.com/newsdisplay.aspx?newspage=15967. Aug. 2008.

[28] Ghostwire.com. *[AS3] Serializing A Bundle Of Bitmaps As Data Objects*. http://www.ghostwire.com/blog/archives/as3-serializing-a-bundle-of-bitmaps-as-data-objects/. Dec. 2009.

[29] Ghostwire.com. *[AS3] Serializing Bitmaps (Storing BitmapData As Raw Binary/ByteArray)*. http://www.ghostwire.com/blog/archives/as3-serializing-bitmaps-storing-bitmapdata-as-raw-binarybytearray/. Dec. 2009.

[30] GM2D. *Simple Flash game in HaXe*. http://gm2d.com/2009/02/simple-flash-game-in-haxe/. Feb. 2009.

[31] Kevin Goldsmith. *CPU, GPU, multi-core*. http://blogs.adobe.com/kevin.goldsmith/2008/09/cpu_gpu_multico.html. Sept. 2008.

[32] Branden Hall. *Understanding Adobe Alchemy*. http://www.automatastudios.com/2008/11/21/understanding-adobe-alchemy/. Nov. 2008.

[33] Ralph Hauwert. *Adobe Alchemy: is it actionscript heresy ?* http://www.unitzeroone.com/blog/2008/11/28/adobe-alchemy-is-it-actionscript-heresy/. Nov. 2008.

[34] Ralph Hauwert. *Another scream on Flash, Alchemy Memory and compilers*. http://www.unitzeroone.com/blog/2009/05/22/another-scream-on-flash-alchemy-memory-and-compilers/. May 2009.

[35] Ralph Hauwert. *Flash 10, Massive amounts of 3D particles with Alchemy*. http://www.unitzeroone.com/blog/2009/03/18/flash-10-massive-amounts-of-3d-particles-with-alchemy-source-included/. Mar. 2009.

[36] HaXe.org. *Why use HaXe*. http://haxe.org/doc/why.

[37] W. Lewis Johnson, Hannes Vilhjalmsson, and Stacy Marsella. "Serious games for language learning: How much game, how much AI". In: (2005). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.97.3050.

[38] Kirupa. *Backface Culling*. http://www.kirupa.com/developer/actionscript/backface_culling.htm.

[39] Jens Krause. *Speed up JPEG encoding using Alchemy*. http://www.websector.de/blog/2009/06/21/speed-up-jpeg-encoding-using-alchemy/. July 2009.

[40] Adobe Labs. *Alchemy*. http://labs.adobe.com/technologies/alchemy/.

[41] Adobe Labs. *Alchemy: Getting Started*. http://labs.adobe.com/wiki/index.php/Alchemy:Documentation:Getting_Started. Nov. 2008.

[42] SegFault Labs. *Asynchronous jpeg encoding*. http://segfaultlabs.com/devlogs/alchemy-asynchronous-jpeg-encoding. May 2009.

[43] LemLinh. *4 Pixel Bender project*. http://www.lemlinh.com/flash-source-4-pixel-bender-project/.

[44] David Lenaerts. *Some Flash Pixel Bender performance tips + benchmarks*. http://www.derschmale.com/2009/07/23/some-flash-pixel-bender-performance-tips-benchmarks/. July 2009.

[45] David Lenaerts. *Some Flash Pixel Bender performance tips + benchmarks*. http://www.derschmale.com/2009/07/23/some-flash-pixel-bender-performance-tips-benchmarks/. July 2009.

[46] Shane McCartney. *AS3 SWF Profiler*. http://www.lostinactionscript.com/blog/index.php/2008/10/06/as3-swf-profiler/. Oct. 2008.

[47] Shane McCartney. *Tips on how to write efficient AS3*. http://www.lostinactionscript.com/blog/index.php/2008/09/28/tips-on-how-to-write-efficient-as3/. Sept. 2008.

[48] Shane McCartney. *Tips on how to write efficient AS3 - Part 2*. http://www.lostinactionscript.com/blog/index.php/2008/11/01/tips-on-how-to-write-efficient-as3-part-2/. Nov. 2008.

[49] Marshall McLuhan. *Understanding media: The extensions of man*. New York: McGraw-Hill, 1964, pp. vii, 359.

[50] Colin Moock. *Essential actionscript 3.0*. O'Reilly, 2007. ISBN: 0596526946.

[51] Nathan. *BitmapData, Vectors, ByteArrays and Optimization*. http://webr3.org/blog/haxe/bitmapdata-vectors-bytearrays-and-optimization/. June 2009.

[52] Nathan. *Optimized Flash Player 10 Z-Sorting Class*. http://webr3.org/blog/flash-10/optimized-flash-player-10-z-sorting-class/. Sept. 2009.

[53] Armand Niculescu. *AS3 Performance Optimization*. http://www.richnetapps.com/as3-performance-optimization/. Nov. 2008.

[54] Chris Nuuja. *Flash Player 10.1 hardware acceleration for video and graphics*. http://www.adobe.com/devnet/flashplayer/articles/fplayer10.1_hardware_acceleration.html. Nov. 2009.

[55] Ted Patrick. *The ABC's of AMF*. http://onflash.org/ted/2007/11/abcs-of-amf.php. Nov. 2007.

[56] Peter Peerdeman. *Intelligent Tutoring in Educational Games*. Amsterdam, 2010.

[57] Pradeek. *Adobe Alchemy: Anatomy of a Simple C Program*. http://pradeek.blogspot.com/2009/06/getting-started-with-alchemy-hello.html. June 2009.

[58] PureMVC. *the PureMVC framework*. http://puremvc.org/.

[59] Quasimondo. *How to draw anything into a BitmapData properly*. http://www.quasimondo.com/archives/000670.php. June 2008.

[60] Paul Robertson. *ActionScript BitmapData and Filters resources*. http://probertson.com/articles/2005/11/02/actionscript-bitmapdata-and-filters-resources/. Nov. 2005.

[61] Rozengain. *Some ActionScript 3.0 Optimizations*. http://www.rozengain.com/blog/2007/05/01/some-actionscript-30-optimizations/. May 2007.

[62] Rozengain. *Using an Alchemy generated texture on a 3D object*. http://www.rozengain.com/blog/2009/04/02/using-an-alchemy-generated-texture-on-a-3d-object/. Apr. 2009.

[63] Jason Shaw. *Adobe Alchemy: Anatomy of a Simple C Program*. http://jasonbshaw.com/?p=158. July 2009.

[64] Jason Shaw. *Getting the Alchemy memory ByteArray*. http://jasonbshaw.com/?p=183. July 2009.

[65] Norm Soule. *PixelBlitz Render Engine*. http://www.blog.crittercreative.com/?p=59. Aug. 2008.

[66] Soulwire.co.uk. *BitmapData Average Colours*. http://blog.soulwire.co.uk/flash/actionscript-3/extract-average-colours-from-bitmapdata. Oct. 2008.

[67] Squize. *Run DMC*. http://blog.gamingyourway.com/PermaLink,guid,b60aaf4d-c94b-4e4a-b7cf-cadf9fb0c74e.aspx. Mar. 2010.

[68] StackOverflow. *Actionscript 3 vs haXe: Which to chose for new Flash project?* http://stackoverflow.com/questions/1044779/actionscript-3-vs-haxe-which-to-chose-for-new-flash-project. June 2009.

[69] Ryan Stewart. *Adobe TV: Double feature on Alchemy*. http://tv.adobe.com/watch/adc-presents/double-feature-on-alchemy/. Jan. 2009.

[70] Tinic Uro. *Adobe Pixel Bender in Flash Player 10 Beta*. http://www.kaourantin.net/2008/05/adobe-pixel-bender-in-flash-player-10.html. May 2008.

[71] Tinic Uro. *Pixel Bender .pbj files*. http://www.kaourantin.net/2008/09/pixel-bender-pbj-files.html. Sept. 2008.

[72] Tinic Uro. *Using BitmapData.setVector for better performance*. http://theflashblog.com/?p=1397. Sept. 2009.

[73] Tinic Uro. *What does GPU acceleration mean?* http://www.kaourantin.net/2008/05/what-does-gpu-acceleration-mean.html. May 2008.

[74] Bernard Visscher. *Using ByteArrays in Actionscript and Alchemy*. http://blog.debit.nl/?p=79. Mar. 2009.

[75] Michael James Williams. *Blitting and caching movie clips in Flash*. http://www.adobe.com/devnet/flash/articles/blitting_mc.html. Feb. 2010.

[76] Justin Windle. *BitmapData Average Colours*. http://blog.soulwire.co.uk/flash/actionscript-3/extract-average-colours-from-bitmapdata. Oct. 2008.

[77] Justin Windle. *BitmapData Colour Palette*. http://blog.soulwire.co.uk/code/actionscript-3/colourutils-bitmapdata-extract-colour-palette. Oct. 2008.

[78] Eugene Zatepyakin. *FluidSolverHD [Alchemy version]*. http://blog.inspirit.ru/?p=339. Sept. 2009.