Juho Ojala

Game Prototype Analysis

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

13.11.2013

Helsinki Metropolia University of Applied Sciences

This thesis includes some technical terminology. The meaning of each term is described in the list below.

Bug	When something in the game is not working as designed or intended.
Feature	A gameplay element, enabling some in-game functionality for the player.
Fix	The reparation of a bug is called a fix.
Green light	Project passes a certain test-plan or quality check.
Hack	A temporary fix or feature on code that is usually made in a rush because a long term solution would take too long to implement at that moment.
Hotfix	A very quick fix that is a must-do, this term is usually referred to when deploying a quick fix to live environment that is already used by a client.
No-go	No-go is a project, which is not seen as worthwhile and will be dropped.
Red light	Project does not pass a certain test-plan or quality check.

These terms are used throughout the thesis as they represent some of the core terminology in the software development field of study.

Author(s) Title Number of Pages Date	Juho Ojala Game Prototype Analysis 27 pages + 6 appendices 28 October 2013
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Miikka Mäki-Uuro, Senior Lecturer Jaakko Haapasalo, Head of the studio

The thesis investigated different production ways of game prototypes and analyses them using an actual game prototype, by codename “Breaker” as a reference. Breaker is produced by Rovio Entertainment Ltd and this thesis contains several confidential sections. These sections are added as appendices and marked as confidential.

The thesis discovers different project management methods suitable for game prototyping and debates between the differences that these methods have, while having the main focus on the methods used in the Breaker. The thesis investigates the pitfalls and general problems that always go together with prototype development, both related to design and programming, having higher emphasis on the programming side. The Breaker code and design will be broken down and analysed for possible improvements in the decision making flow.

The author worked as a programmer in the prototype development team during the two month prototyping stage. Along the author, three other people worked on the project as a programmer, a game designer and an artist respectively.

Keywords	Rovio, Breaker, game prototype, prototyping, project management, design, programming

Contents

Appendices

1 Introduction

Developing a game differs a lot from developing other information technology software. While the gaming industry has roughly the same production stages as other software: concept, prototyping, preproduction, production and maintenance, the iteration of these stages functions a bit differently for gaming. The main reason for this is because the game has to feel fun, addictive, visually appealing, rewarding and challenging in the right scale. While keeping these goals in mind the game also has to fill the same requirements as any software, it must run stable and bug free. While the concept of running stable is relatively straightforward, it is hard to measure factors such as fun. This is why the prototyping stage is such a crucial part of game development.

The game development usually starts from an idea, a concept that somebody thought might make a fun game. During the prototyping stage, this concept will be made into a game with most of the core functionality present, to test and see if it actually is fun in reality. The common verdict is that something feels a bit off, the fun is not there, the goal of the game is not clear and a lot of other details just do not work. This is when the development team starts to think how to improve, or even completely change these parts of the game and create a new iteration of the game. This iteration process is recursively completed until the development team feels comfortable that the game is fun, the game works and the core gameplay is good.

Rovio Entertainment Ltd produced a game prototype for a project called Breaker. The development of the Breaker is followed and referenced in the confidential appendices of the thesis.

The purpose of the thesis is to find the essence of game prototyping, analyse different project management methods, programming methods and design decisions that can be used and also the ones that should be avoided. The Breaker will be a real life example that will showcase many of these topics in action. After reading the thesis, the reader should have comprehensive understanding of game prototype development.

The study is structured so that first it goes through the general production flow of the game. After that the project management of the game product is opened in detail. Followed by an explanation of designer work and how a designer works specifically in a

prototype stage game. The last topic is the vastest, explaining programming work and necessary tools required for it and discovering the choice between a hack and quality code. The study finishes up in discussion and conclusions, which highlight some of the author's opinions on these topics and sum up, what has been learned throughout the study. The thesis also contains confidential appendices, which go through and analyse the development cycle of the Breaker prototype.

2 Game Development

The purpose of this section is to give a general idea of how each production stage of game development cycle works. Something to keep in mind while reading this is that these production stages are somewhat loosely defined and some people would argue differently where to draw a line between preproduction and production, or whether prototyping is part of the preproduction, or if release candidate is the same thing as code release or not.

2.1 Concept

Concept is the basic idea for the game. A good concept creates a good base for a good game. The concept stage is very loosely defined as sometimes the concept can only be a single picture or a phrase. Usually however the concept will already include some design, some pictures and description of the game as the creator of the concept visualizes it. It is very important that the concept is intriguing, but it is rarely the concept that is faulty if the game does not succeed. See the Breaker project concept in Appendix 1 (confidential).

2.2 Prototyping

The prototyping phase is all about taking the concept and making a game around it. This phase of the game development is absolutely crucial for the success of the game. The game does not need to be completed, but the main features should have been founded. After the prototype the goal is not to have a final game, but the player should easily be able to identify what is fun in this game and what is the goal of the game. The decision between proceeding with the project or a no-go has to be made at the end of the prototype phase. See what kind of prototype goals were set for Breaker in Appendix 2 (confidential).

2.3 Preproduction

This is the phase where the team is supposed to clean up after the prototype. Not only the code but also some parts of the design are usually in a very chaotic state after prototyping. Preproduction should be the time to clean things up and prepare for the production phase.

2.4 Production

In the production phase it is still possible to add minor features to the game, as well as polishing the major ones. Game polish will take most of the time along with bug fixing. There are several important milestones in the production phase: alpha, beta, code release / release candidate, gold master. When the project is approved to move to alpha, it is assumed that all the features are there in the game already, it is still acceptable to do minor polish on those but even the minor features should already exist at this point. Assets on the other hand do not need to be finished for alpha. Beta is the next milestone after alpha, in beta the game should be completely ready except for bugs. After entering beta, it is assumed that the only thing left to do is to fix the bugs, that means all the logic, features, assets and polishing should have been done by now. Code release or a release candidate is a build of the game where the team feels that the game is ready to be released. The quality assurance team tests the release candidate according to a test-plan and either green lights or red lights it. Usually it takes several candidate builds before it gets green lighted. Gold master is the final milestone, this is the final build where everything is done, tested and accepted. Accepted gold master is basically the final game.

2.5 Maintenance

After the game is live and masses of people start to play it, new bugs are usually found. Part of the maintenance job is to hot-fix the most critical bugs, the minor ones are more likely to be addressed in the next update. There are basically two kinds of games that have long lifetime: the type that gets frequently updated with new features to keep players interested and the type which has an endless game cycle. Obviously the latter requires less maintenance work and is usually preferred for a lot of games. However, most games just do not work with this kind of design and that is why the development team will usually start to make an update for the game after gold master has been

achieved. Updating of the game usually is continued for as long as the company management sees it profitable.

Game development consists of the five main phases described above. The first phase is the concept which is the basic idea for the game. The second phase is prototyping where the development team tries to find the fun part of game. The third phase is preproduction where the code and design are cleaned from prototype phase hacks. The fourth phase is production, which has four important milestones: alpha, beta, release candidate and gold master. In the production phase the game will be added with remaining features, polished and finished. The final phase is the maintenance where the development team will continue to make more content to the game as well as fix existing bugs.

3 Prototype Project Management

This chapter goes through the management part of the project. Paragraph 3.1 oversees what kind of a role is responsible for the task. After that, the study proceeds to see what kind of tools can be used and finally discusses the different management methods.

3.1 General Role

There is usually one person designated for project management. This position can have different names depending on the managing methods and company terminology. Some of the more known titles for this position are project manager, scrum master and most commonly in game industry, a producer. This position usually comes also with other responsibilities as well. The producer is not only the person who is in charge of the organization of the project, but also is responsible for the project. This means that the producer has the power and the responsibility to have the last say in every decision. The producer also takes care of the team, gets rid of other hindering problems and allows the development team focus completely on the project itself. Sometimes if the project is large the responsibility is split between the team management and the product responsibility. These positions in game development are each respectively called producer/project manager/scrum master and executive producer/product owner. During the prototype phase there will usually be times when uncertainty of suitable solutions is in the air and the designer and programmer might turn to the producer for the

final word. For these cases it is vital that the producer knows just a little bit of both of these areas.

3.2 Methods

There are several popular project management methods, but the ones that have landed most successfully for software development are scrum and waterfall. Scrum is arguably the most common software development method. The idea in this method is simple, in a simplified example there are three kinds of people: product owner, scrum master and developers. Product owner is the customer who needs a certain kind of software or an executive producer who has responsibility of the product. Product owner will specify the requirements for the software and can also affect the decisions regarding the order of developed features. The scrum master is the person who organizes the project for developers. The Scrum master works as a buffer between the team and everything else. Together with developers the scrum master will split the software to different features. This list of features is called the backlog. Implementation of the backlog features will be split to smaller tasks, but usually at a later stage. In an optimal situation developers only need to worry about completing the tasks. One of the most important characteristic of the scrum development is that the tasks should not be dependant of each other. This may be impossible to achieve at times but it is the ideal that should be chased. The work is done in sprints, where a sprint represents a short timeline usually between one week and one month. After each sprint the product should be in usable state. The sprint starts by deciding on sprint goals, what are the things the team wants to implement during this sprint. These will be divided to tasks if not done so yet and assigned to appropriate developers, who estimate the length of the tasks. The workload of the task is estimated in man-days of effort, for example 5 man-days of effort can be 5 days of a single person working or 1 day of 5 people working. After the sprint plan has been made and the goals have been agreed on the team starts to work on the sprint. During the sprint no contact is usually kept with the product owner but the team will arrange frequent meetings to check each ones progress on their tasks. At the end of scrum sprints there is often a retrospective, where the team discuss what went well and what could have been improved, how to have better tackled problems they had hard time with and how can they avoid these problems in future. After several sprints by the team it is possible to estimate the team's progress velocity which is basically the relation between the time estimated by the team for the tasks, and the time that it actually took them to complete the tasks. The velocity is a very use-

ful piece of information for the scrum master as well as the company management. See Figure 1 below for a graphical presentation of the scrum system.

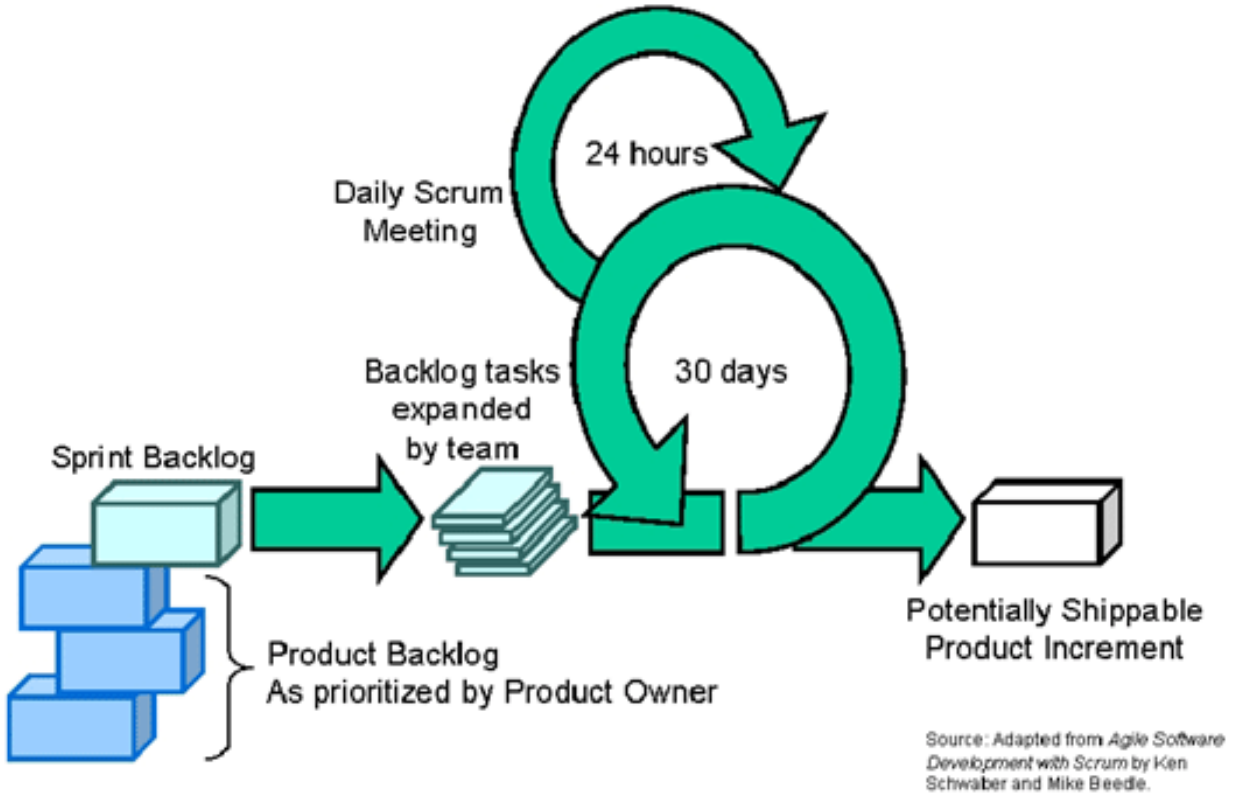

Figure 1: Scrum model [Reference 2]

As seen in Figure 1 the blue blocks compose the sprint backlog of which team will then take sprint tasks. The sprint duration in this figure is 30 days and a daily meeting is held every 24 hours. A functional product is always the end result of the sprint.

The waterfall model implies that the tasks of the project are to be completed in certain order. Starting with specifying the requirements and followed by design, implementation, integration, testing, installation and ending with maintenance. The waterfall model works well in micro-cycles of game development. Micro-cycle in this case means individual features instead of the overall process. A lot of software companies have given up totally on waterfall and moved to the scrum or other agile development system, but in games development waterfall still holds some ground.

See Figure 2, for a graphical presentation of a simplified waterfall model.

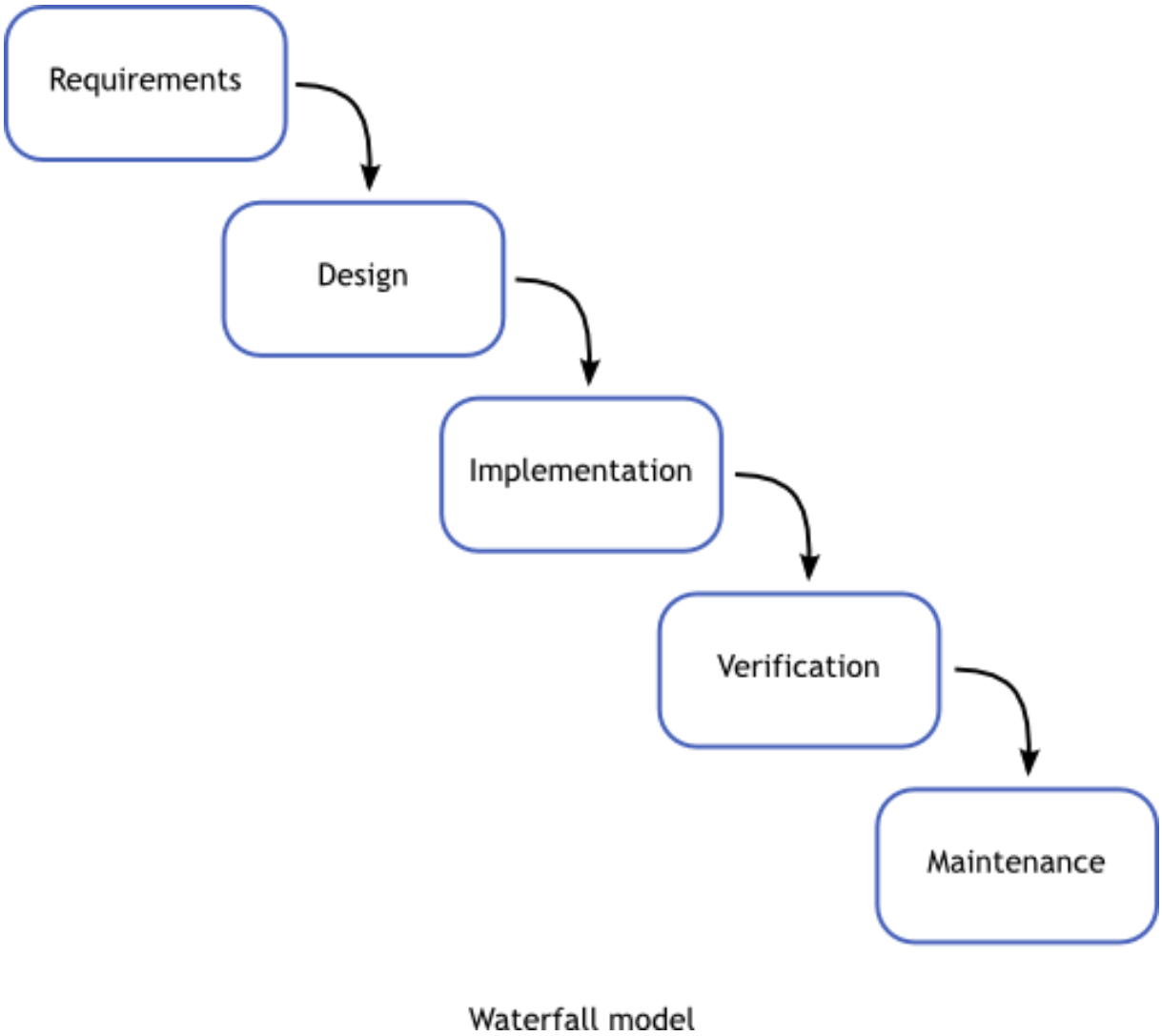

Figure 2: Waterfall model [Reference 3]

This figure is a slightly simplified version of the waterfall model discussed earlier, but the idea is the same. First the requirements are defined and after that design, followed by implementation, verification and maintenance. The important thing in the waterfall model is that the previous section is completed before moving to the next one.

3.3 Tools

There are a lot of tools available for all the different phases of development as well as different tools for different project parts. It is best to use as few tools as possible. This

does not mean one should avoid using project tools, it just means that one should try to centralize the project tools as much as possible. One of the reasons being, that it reduces error proneness. Also it makes the life of the developer a lot easier if one only needs to check one place for tasks or bugs. There are several tools for the job. The simplest one consists of a whiteboard and some post-it notes. One simply writes a task on a post-it, write each team members name on the whiteboard and assign the notes to the people designated to the task. There can also be a completed section where the notes will be moved after the developer feels that the task is done. This can be a quite effective tool in a small group and is usually associated to the scrum-management method. This classic scrum way of handling things comes with a lot of limitations, e.g. if the team consists of more than ten people, the whiteboard will run very rapidly out of space. See Appendix 3 (confidential) for project management tools and methods that were used in the Breaker.

4 Game Design

This section covers the role of the game designer and more specifically the designer's work for the prototype stage. See the design flow and analysis on it for Breaker in Appendix 4 (confidential).

4.1 General Role

To put it briefly, the role of the game designer is to design the content and rules of the game. This is a very vast definition and divides into several major design roles such as content designer, mechanics designer, economy designer, monetization designer, storyline designer, user-interface designer and control designer to name a few. Depending on the size of the project these can be all done by one person, or be split into even smaller categories for several people per each design job. The position of the designer is very difficult in the way that designer position requires understanding of programming, producing, modelling, animating, arts and pretty much every area of the project. The designer should also be able to fluently communicate the features to the other developers, usually the best way is some kind of mock-up as otherwise the thought relies on the fact if the other person understood the description in the same way as the designer. For example, trying to describe a graphical user-interface without drawing a mock-up will one hundred per cent be different from the original thought. The tools of the designer should be those that the designer feels fitting, usually including

drawing tools e.g. Adobe Photoshop. The design work requires a creative mind to think of intriguing ideas and a solid understanding of implementation mechanics to know, which features consume a lot of time to implement. The designer must also think about the player experience and keep the different features closely tied together. There have been a countless number of games where designers were too eager to put in too many different features, not polishing any of them or not tying any of them together. The features not tied together often feel unrelated and might break the game into too many different directions.

4.2 Prototype Design

In the prototype phase the designer is usually already aware of the concept. This means spending some time to discuss with the concept creator and brainstorm with the development team for good ideas. The most important goal for the designer in the prototype phase is to find the core gameplay. This is difficult even if the concept is good, because the core mechanic should be the thing that feels fun and brings the player back to game. A good way to start a research is by looking up some similarly themed games for references. Playing these games usually brings out a lot of new ideas as well as opinions on what feels bad or good in each of these games. It is very likely that the designer will come up with several ideas which one has to weight and think through. After one is set to an idea one has to clean into an understandable and clear version. With this being done the backlog should be created together with the development team, producer and the product owner. After the core idea is waiting for implementation by other developers there is a lot of design still to be done regarding the user-interface, balance and all the other issues, as well as polishing the core concept.

5 Programming Game Prototype

This section covers the role of the programmer, tools that can be considered for programming prototypes and some decisions that the programmer needs to make.

5.1 General Role

Programmer is the muscle of the group, the person who brings the art and design together into a game. Besides from stating the obvious which is that the programmer writes all the game code, it is also the programmer's responsibility to bring technical

perspective and limitations to the design discussions. For example a designer wants a cool new animation feature for a creature in the game. The programmer knows that it is not possible to put it in with the currently integrated animation system. It is the programmer's duty to bring this point out and make an estimate on how long it would take to modify the animation system so that it could be used for the new animations. A programmer is also responsible for the quality of the code. The code should be solid, flexible and informative. Solid means that it is not easily broken in corner cases or at integration of other systems. Flexible means that the code is structured smartly in a way where it is easy to modify any of the implementation layers without having to rewrite the whole thing. Informative means that the code is very clear and packs the necessary information for other programmers to pick it up from where it is been left. This can be accomplished by writing comments but even more important than comments are smartly named variables and smartly picked variable types. See Figure 3 for a bad example of naming and typing.

```
//p is the number of parts
int p = 0;

for (std::vector<int>::iterator it = parts.begin() ; it != parts.end(); ++it)
{
    ++p;
}

//this is safe because p is always positive
float s = sqrt((float) p);
```

Figure 3: Bad naming and types

As we can see in the figure, due to bad naming and type of p the comments are used to fill in the missing information.

See Figure 4 below for a better example in naming and types.

```
unsigned int partCount = 0;

for (std::vector<int>::iterator it = parts.begin() ; it != parts.end(); ++it)
{
    ++partCount;
}

float s = sqrt((float) partCount);
```

Figure 4: Good naming and types

Figure 4 explains all the same things without any comments using a type that tells that the variable is always positive and a variable name that already tells one what information this variable contains. Writing the code as in Figure 4 can be called as self-documenting code. One should write comments anytime the code is not understandable for someone else without them, but that just means that the code was not very clear. Comments are usually required mainly for complicated operations which may exceed normal coding expertise. Another quality code aspect is consistency. This is perhaps the most overlooked quality factor in the code. Consistent code is easy to read, easy to understand. See an example of no consistency in Figure 5 below.

```
s = sqrt((float) partCount);

{
    /*Bunch of stuff everywhere in code*/
}

s= 4.f;
```

Figure 5: No consistency

It can be seen in Figure 5 that the second assignation of variable s does not contain the space before the assignation, which it does in the first one. This is inconsistent.

See Figure 6 below for consistency.

```
s = sqrt((float) partCount);

{
    /*Bunch of stuff everywhere in code*/
}

s = 4.f;
```

Figure 6: Consistent code

Figure 6 has same number of spaces used and it is consistent. This may seem like a very minor detail to many but it is actually a rather big deal. For example imagine having a global variable called g_worldScale which represents the scale of the world. The project at hand is big and has several hundred code files. It is known that in some corner case scenario a wrong value will slip into this variable. To find out what is wrong one wants to debug all the places where this variable is being changed. With consistent code one can count on the fact that these places can be found by searching for "g_worldScale =". However if the code is not consistent and there is a different number of empty spaces or different kind of value passing methods to this variable, one will be in trouble trying to find all the places.

There is usually a project lead programmer and in bigger companies even a company or a studio lead programmer. The project lead programmer is in charge of overall quality of the code in the project. Project lead programmer will review code from other programmers by request and is the first person to ask for advice when one is unsure of how to implement something. If there are disagreements on tech choices of the project, the lead programmer has a more powerful say in it than the other coders. The studio and company leads are respectively responsible of the code quality on studio and company level. See Appendix 5 (confidential), for a code analysis in project Breaker.

5.2 Differences of Scripting and Programming

Technically the difference is that scripts are interpreted and programs are executed. What this means in practical terms is that a program is compiled (or otherwise derived) and then run. The compiled program is in executable form which is considered not human readable. An actual program does not need another program to execute it. Scripts however are only interpreted from the actual programs and they need another program

to execute them. A script works independently or externally and can be disabled without aborting the actual program.

5.3 Programming Languages

There is a countless number of different programming languages, yet only few of them are used for developing games. This chapter discusses introduces some of the generally more popular languages and group them up by categories.

5.3.1 Easy and Effective

This section concentrates on two languages: Java and C#. Both of these languages offer powerful tools to create content fast and relatively safe from human-errors. There are a lot of powerful functions and libraries existing and reduced accessibility to lower level functions. This makes it fast and less error-prone but also limits one to using the available features. While the functionalities are made by some real code wizards and are of highest quality, the more the features do and the less error risk they provide means that the more general they are. The more general something is the less feature specific optimization can be done. A good rule of thumb is that general functionality comes at the cost of performance. Same goes for eliminating programmer errors, if one makes automated garbage collection and limited pointer functionality, it comes at a cost as well. Programming a game is very performance driven so if one is afraid of making mistakes by playing with pointers and memory management, one should probably consider honing one's programming skills on these areas over picking an easy language. The platform one is programming to weaves a great impact to the programming language choice. Thus, when programming for an Android device, a Java driven platform, it does make some sense to pick Java as the programming language as well. This stands true only as long as the Android is the only target device. C# is often falsely thought of as a good performance language but it has actually only slightly better performance than Java. The good thing about C# is that there is a very powerful tool that utilizes it well called Unity, more about this tool will be discussed in Chapter 6.4.

5.3.2 Hard and Arduous

As explained in the previous chapter, generality and already implemented functions reduce the chances to do specific optimizations. Does that mean that writing pure machine code would be the most effective? Yes, in theory. Writing only pure machine code would limit the people who can make a modern game that way to only a handful of programmers and even they might do some errors that could lead in worse performance than by using existing functionalities that have been developed over time by the top programmers. Writing a game purely on machine code is possible, but very painful, very time consuming and very error prone.

5.3.3 Happy Medium

There exists a programming language that has very good performance, a lot of powerful functionalities and tolerable amount of error-proneness and that is C/C++. These are two different languages but often coupled together because C++ is according to the developer of the language Barne Stroustrup “a better version of C”. C++ is absolutely overwhelming in the game development industry and if one wants to find a game development company that does not use it, one would probably have best luck looking for small start-up companies who only develop games for single mobile platform or web-based technologies. If one looks purely at the language then C++ is the top contender.

5.3.4 Scripting Languages

There are several scripting languages to go by Python, Lua, JavaScript and Ruby just to name a few popular ones. They all are good in their purpose and one should choose the scripting language by deciding which one is easiest to integrate to the project and easiest for designers to write.

5.4 Tools

This is the area where a storm is emerging right now. A new tool called Unity is gaining more and more ground on the game development industry. This chapter breaks down the pros and cons of Unity development as well as go through other vital for programmer tools.

5.4.1 Unity

Unity is not a typical programmer tool. It is closer to a designer tool. It breaks the way programmers usually do things as one is no longer a master of everything but rather use the super powerful Unity editor to create a scene and simply add scripts to the objects if one wants to provide them with special functionalities. Unity supports programming in three languages C#, JavaScript and Boo. Once a programmer is familiar with Unity he can create a solid prototype even in a single day. The value in the development speed is easy to see, not only that, but the graphics one can do with Unity are fairly impressive as well. See Figure 7 below for the graphical visualization of the scene in Unity.

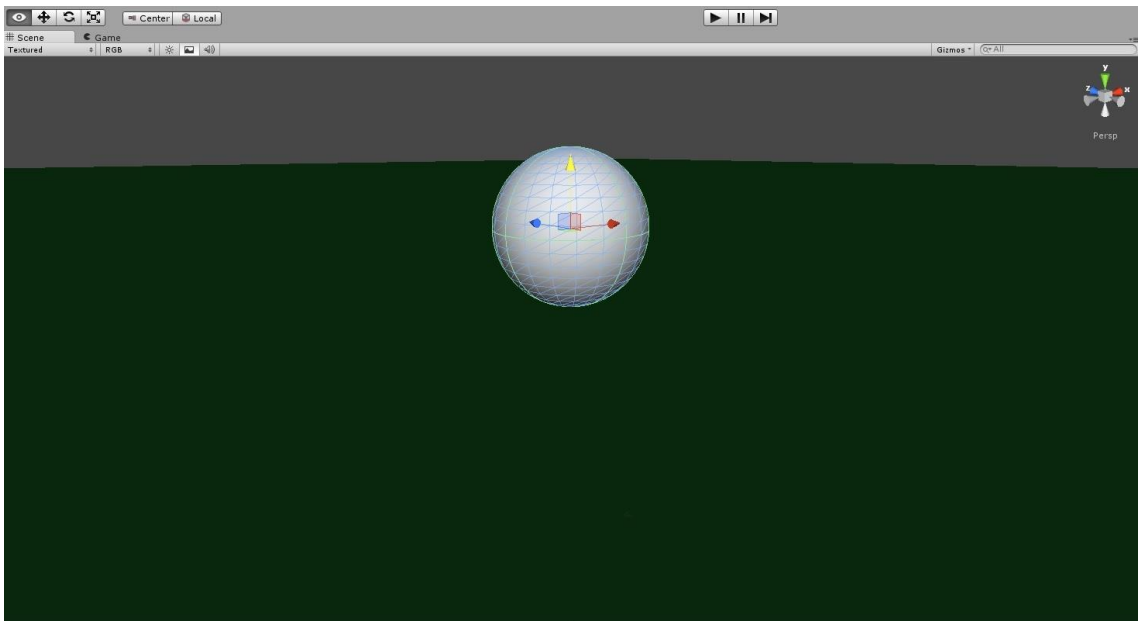

Figure 7: Unity scene

Figure 7 demonstrates translating a sphere inside the Unity scene. Adding geometrical objects or importing 3D models to a game is as easy as drag and drop, adding gravitations is a check-box, script values can be modified from editor side and many others things are unbelievably easy to do.

See Figure 8 below on how easy it is to change object values in Unity inspector.

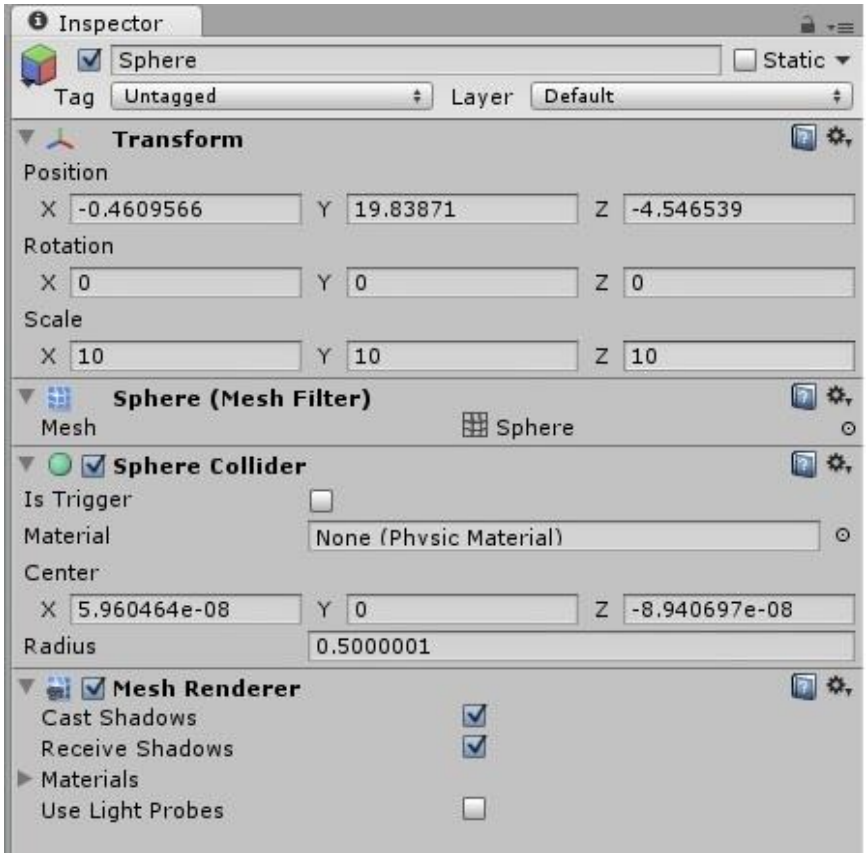

Figure 8: Unity inspector

Figure 8 displays some of the attributes of the sphere in the inspector view. One can see that for example shadows could be removed from this object by simply tapping a check box.

It is easy to see why many people are falling in love with this editor. It also supports multiple platforms which is very handy from the developer's point of view. What is the catch then? As noted already earlier, great power comes with great restrictions. While most of the things are easy to do, there are a lot of things that the programmer cannot change. These things might impact the performance heavily or slightly, depending on the game. Something so general and so powerful cannot simply be as effective as a specifically written code for that game. There is also an issue that one cannot clear the Z-buffer during frame, this could complicate things a bit when creating a split screen game. Also from the programmer perspective it is very annoying that at least for now, one is unable to access stencil buffer in Unity. There are also some issues with version control of Unity projects. Also the 2D-asset pipeline is still under construction for Unity.

The biggest issue still remains in the performance though. Unity is extremely fast for prototyping, but a new problem may occur if used solely for that purpose. Following is a practical example of the issue: Imagine that one is creating a very demanding game performance vice. There are a lot of iterations to go through and developers decide to use Unity during prototyping. The team ends up finding a feasible design and the management wants to put the project into the preproduction phase. There will be performance issues if the team keeps developing it with Unity, but the other option is to build the now already running game from the scratch with a completely different technology. That is not only work one does not want, but most likely a lot of time that was not budgeted for the project. It is important to remember that game development is an entertainment business, a really hard one to succeed at that and time frames are usually extremely tight. This is why it is essential that an experienced programmer makes a judgement call before starting programming the project if it should be developed with Unity or other tools.

5.4.2 Visual Studio, XCode and Eclipse

As stated in chapter 5.3, C++ is often the language of choice when developing games. In the author's experience the best tool to develop C++ with in Windows is Microsoft's Visual Studio. It has very good and flexible tools and a very nice compiler. It is easy to control error levels so one can easily find all the warnings and errors through it. Debugging is easy and smooth and one can even try to debug crashed exes that were not run in the Visual Studio originally. It is also easy to find plug-ins to Visual Studio to customise it even better for personal likings. See Figure 9 below as to the basic layout of Visual Studio.

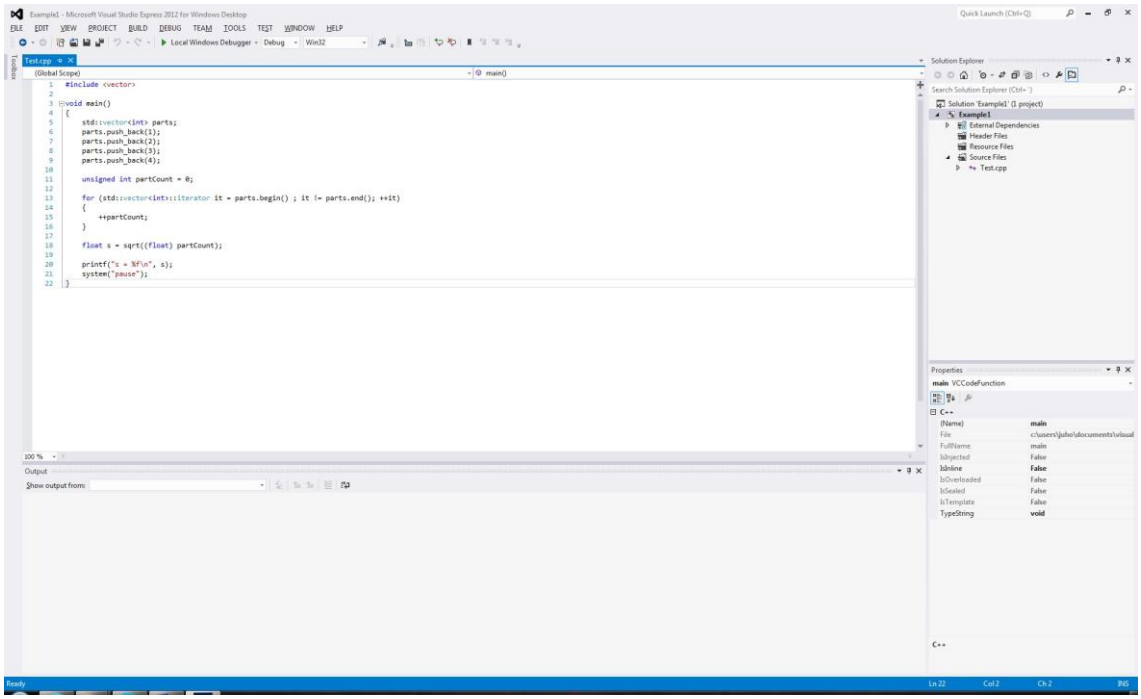

Figure 9: Visual Studio

Figure 9 demonstrates an unconfigured layout of Visual Studio Express 2012, which contains a project view on the right, code window in the top middle and output in the bottom. These can be easily changed if the user wants that.

If one works on OSX, XCode is the top candidate. Just like Visual Studio, XCode has nice basic features and for example debugging from IOS mobile device is extremely easy with XCode. The downsides would be the lack of third party plug-ins, several known bugs and fairly slow. Figure 10 below shows XCode.

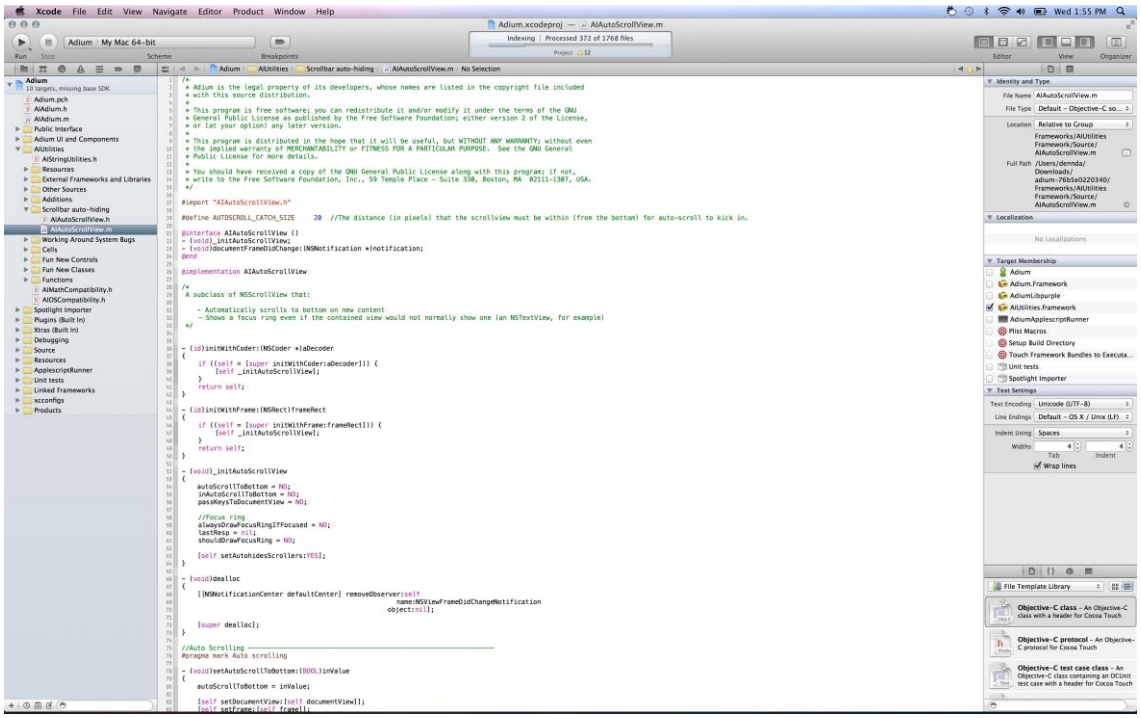

Figure 10: XCode [Reference 5]

Figure 10 displays XCode and it can be seen that the layout is fairly similar, this time the project view is on the left and the code in the middle.

If one really wants to develop on Java, which would most likely mean working on an Android platform (with no plans for multiplatform) then Eclipse is the best tool for the job. Though the debugging is clumsy compared to Visual Studio and XCode, the previous two are not Java development tools and out of Java tools Eclipse has the best synergy with Android by far.

The Eclipse editor can be seen in Figure 11 below.

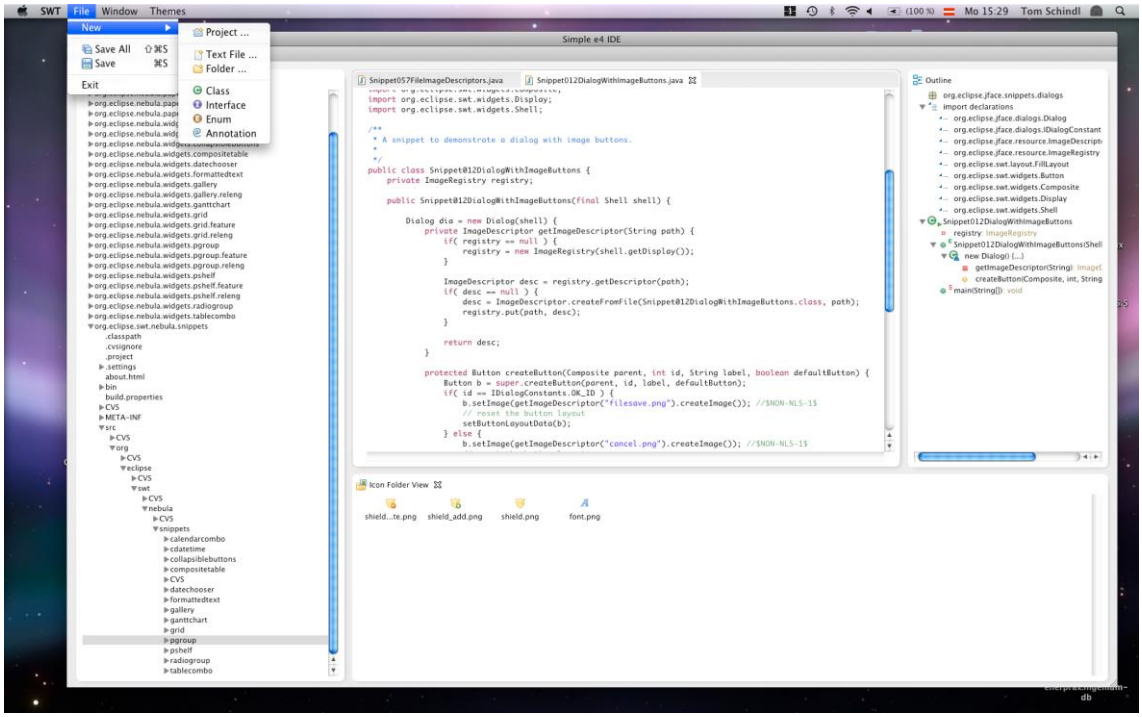

Figure 11: Eclipse [Reference 6]

Figure 11 displays Eclipse, which is very similar to the previous two from layout perspective. The biggest difference between the three comes from functionalities, operating system and the programming language they are best associated with.

5.4.3 Scripting Tools

There are some excellent tools available: Sublime, Notepad++ and Editpad Pro just to name some of the most popular ones. All of these editors provide great tools for searching elements, highlighting language reserved names and functions that make a programmer's life a lot easier.

5.4.4 Version Control

To manage changes in the project in a sensible way one needs version control. It would be very painful to send modifications over mail to each other and manually try to figure out if one is about to destroy somebody else's work. The version control will enable easy revision based project management where it is easy to make changes and notice conflicts when two people are editing the same files. One will be able to merge the conflicts and keep the overall version intact. It is also very easy to pull back and go

to a previous revision. It is easy to create branches and tags which are used to develop a program to certain direction or to hold the repository of the project that was at certain stage (for example 1.0 release for public). The two most popular version control systems are Subversion and Git. Besides slightly different syntax, the main difference between the two is that the Subversion has one central repository where each member of the project will commit to. See Figure 12 below for the basic functionality of Subversion.

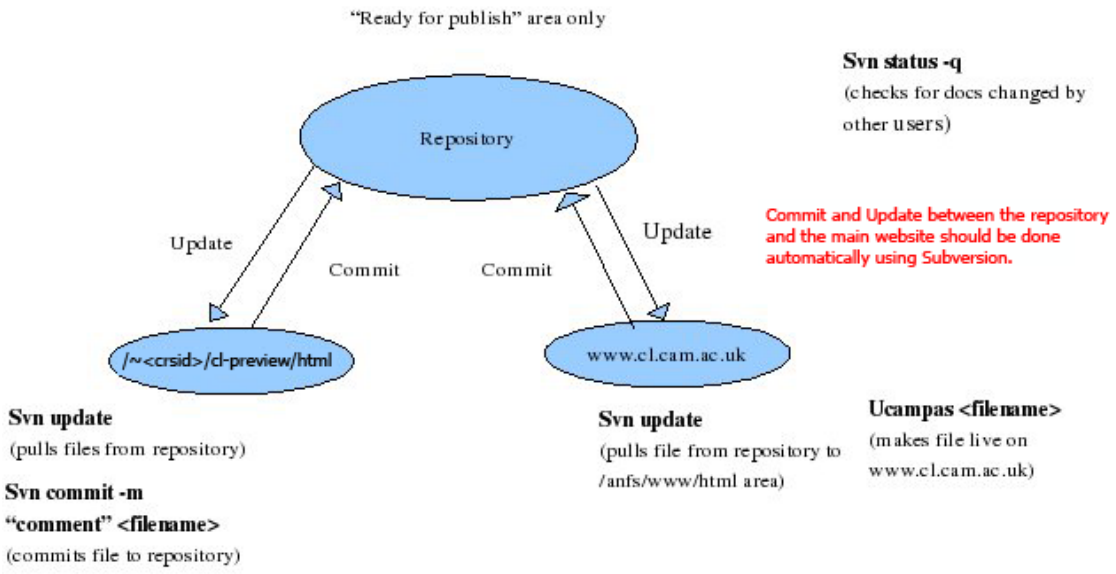

Figure 12: Subversion usage [Reference 8]

Figure 12 displays an example where two different clients are modifying the same code base. The update function will update the local repository of the user to the newest version and commit will push the local modifications to the server so that the other clients can have them with update.

Meanwhile Git has both a local repository and central repository so the user can track local changes as well. See the basic usage flow of Git repository in Figure 13 below.

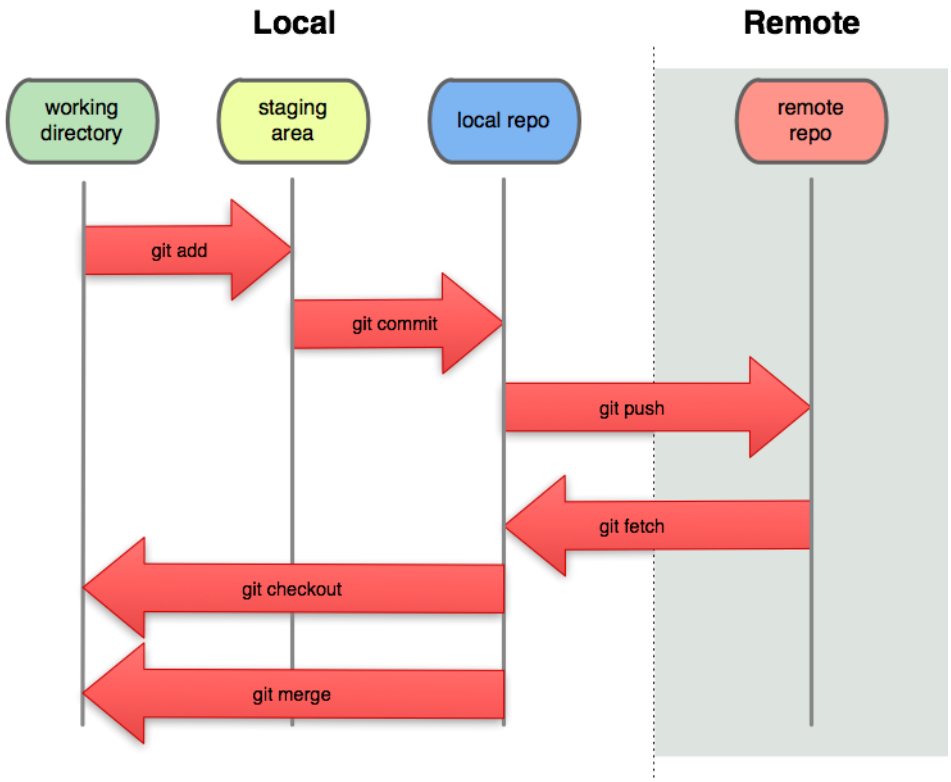

Figure 13: Git usage [Reference 9]

Figure 13 displays how Git has more layers. First the changes made in the working directory are added to the staging area. After that the changes are committed to the local repository. From the local repository the changes are pushed to the server. Changes made by others can be fetched from the server to the local repository.

Many people claim that Subversion is much simpler to understand and use, which can be a benefit if the team has a lot of non-IT proficient members, artists for example. On the other hand Git is a lot faster than Subversion, which can be a considerable benefit when working on huge projects. Both systems are widely used and accepted. There are several tools to make the usage easier, the most widely approved probably being Tortoise, which adds a graphical UI to use both of these tools. Tortoise is also really easy to understand and has great merging tools.

See Figure 14 below for how Tortoise adds options when right clicking on folder in Windows.

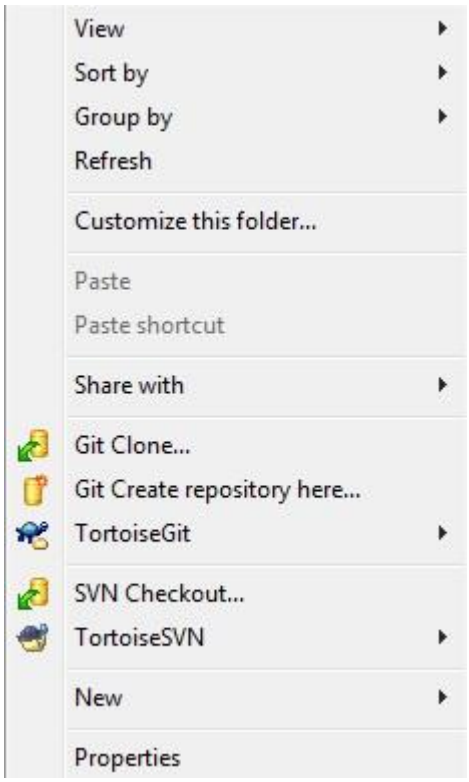

Figure 14: Tortoise

Tortoise adds many options to the right click menu on Windows, which can be found under the TortoiseSVN and TortoiseGit labels.

5.5 Build Automation

To put it simply, building is about compiling the source code, linking the object files, running tests, copying assets and depending on the specific project doing other necessary steps to create a runnable game or installable package from the repository. Though the build system may not seem that necessary in very small project, when one starts to work it can be very shortly seen how it becomes a core part of the loop. Automated building is very useful for example in Scrum system where a solid state of build is required every end of sprint. With integrated testing one can also eliminate a lot of error factors. Moreover, as a project grows bigger and one starts to add third party libraries, external modules and other components together the building process can become rather complex. In those cases one would appreciate the already once set-up

automated building that takes care of everything, instead of spending hours trying to figure out the correct building order oneself. One of the most popular tools to use for build automation is Jenkins.

5.6 To Hack or Not to Hack

While prototyping under a tight timetable, one can expect to face the decision of hacking daily. A hack is generally speaking a bad thing, temporary solution that often has no solidity and no flexibility. Why would one make a hack in the first place? It all comes down to time and quality ratio. Sometimes hacks can save incredible amounts of time. Consider the following simplified example scenario: the team wants to try if the game becomes more fun by adding some real-life like physics. One can either start to implement a very arduous and time consuming physics engine or one can create fake physics by translating objects in a similar way. A good physics engine implementation might take months, while faking them might be done within a day. For example, the development team has been budgeted for three months to make something good out of the proto concept. Making the physics engine seems hardly a solution for only testing it out. A hack may indeed be the best solution to the problem. Then again, hacking it now might bite back later if it was decided that this feature is fun and will be part of the gameplay. It is smart to create the system as flexible as possible so that one part of the game can be changed without major code rework. There is often a straight correlation between non-flexibility and the number of hacks. Hacks should be avoided at all costs if the schedule allows it. One could say they are a tool against deadlines. Tight schedules will most of the time result in a drop of quality.

6 Discussion and Conclusions

There are five development phases to a game. The prototype development phase of is the most rapid one, constantly making changes to all parts of the game. The goal of the prototype phase is to find the fun inside the game. Finding the fun usually takes several tries and tends to leave lots of mess behind from the changes. The preproduction phase is used to clean the mess left by the prototyping phase. The production phase will add the minor gameplay features not yet implemented, polish the existing features and fix all bugs. During the maintenance phase the game is updated. The prototype

phase is the most crucial, but a good development team can bring a lot of life to even a badly prototyped game with good polish and meaningful meta-game.

The producer, or in some cases the executive producer, is responsible for the product quality and meeting the deadlines. Producer will handle the project management and tries to help the development team with any problems hindering their work. To handle the project a project management method needs to be used. This is most commonly the modern and very agile Scrum. The other good option is the very strict waterfall method, which suits game development rather well too. The author's preference is practical project management. Scrum has a lot of good sides to it, but sometimes it is a bit too optimistic and naive. A system where the overall process is being handled via the scrum approach, but the actual tasks are tackled using the waterfall model can be extremely efficient. A tool is also required to handle the project management efficiently. Some might use only whiteboard, but the use of one of the digital tools is strongly recommended. There are quite a few good ones available.

The designer's main job is to design the gameplay and balance it out. The designer is mainly responsible for making the concept into an intriguing gameplay.

The programmer's job is to implement the designer's vision as faithfully as possible, trying to keep the code quality as high as possible. Some people might also consider getting things done in a certain time a quality, even though the actual code might be very hacky. It would be recommended to keep things simple and call flexible and consistent code as quality. If the timeframe is short it just means that the code quality will drop.

During the prototype phase there are often difficult decisions to be made which can cost dearly at the later stages. These decisions are usually about choosing the right tool or making a choice between time and flexibility. Flexible solutions are necessary to be able to quickly change the game without redoing everything, but usually the time is also heavily limited. These decisions should be thought over together with the designer, programmer and the producer attending. Many of the decisions are greatly influenced by experience. If lacking experience, it is a very good idea to consult a senior employee of same field.

The project lead programmer needs to choose the technical tools and programming languages for the project. C++ is the cookie-cutter language for game programming and for scripting Lua is good as it is easy to integrate with C++ code and is also favoured by most game development companies. Visual Studio is the program one would recommend for C++ and for scripting there are a lot of efficient software available. The author has found Notepad++ to fulfil all requirements. Unity is also a really good tool for prototyping, but comes at a performance cost. Unity has three languages available: JavaScript, Boo and C#. C# is recommended as it is the only true programming language out of the three, the other two are scripting languages.

Build automation makes life easier for quick building, testing, keeping track of the product integrity and distributing the builds to the right people. The most commonly used build automation tool is Jenkins, if Ant and CMake are not considered tools at their own right.

See the Breaker project conclusion in Appendix 6: Breaker project conclusion (confidential).

References

1 Scrum (software development). Web document. <http://en.wikipedia.org/wiki/Scrum_(software_development)>

2 Scrum model picture. Web document. <http://www.methodsandtools.com/archive/scrum1.gif>

3 Waterfall model picture. Web document. <http://duncanpierce.org/files/images/Waterfall_model.png>

4 Scripting languages. Web document. <http://en.wikipedia.org/wiki/Scripting_languages>

5 XCode picture. Web document. <http://denter.org/media/images/xcode_retina/xcode_retina_1.png>

6 Eclipse picture. Web document. <http://tomsondev.files.wordpress.com/2010/05/screen3.png>

7 Revision control. Web document. < http://en.wikipedia.org/wiki/Revision_control>

8 Subversion picture. Web document. <http://www.cl.cam.ac.uk/local/web/subversion/introduction/svnHousekeeping.png>

9 Git picture. Web document. <http://thkoch2001.github.io/whygitisbetter/images/local-remote.png>