# The Game Asset Pipeline

**Joost van Dongen**

*August 6th, 2007*


**Supervisor: Jeroen van Mastrigt**

2006-2007
EMMA Game Design & Development
Utrecht School of the Arts

**Abstract**
The game asset pipeline is the path that all models, textures, sound effects, levels, animations and other assets follow to go from the tool in which they were created to the actual game. This path can include exporting, optimising, checking for correctness and pre-calculating data, but also the content management system that stores all the data and keeps track of versions. This Master's thesis analyses the aspects of the game asset pipeline and their relation to each other and to the game design and development process.

# Contents

# 1 Introduction

The game asset pipeline can make or break the development of any game. It can double the production time of a new game if done badly, or, if done well, make the tweaking of all the little details that are needed for a polished product a joy for the asset developers. The game asset pipeline can frustrate your team into looking for a job at the competitor, or improve the framerate of the game by automatically optimising assets.

The claims above might seem a tad strong, but the way the game asset pipeline is laid out can indeed have these effects. So what are we talking about exactly? In his article in Game Developer Magazine, Noel Llopis defines game assets as follows:

"*Game assets include everything that is not code: models, textures, materials, sounds, animations, cinematics, scripts, etc*." [Llopis2004]

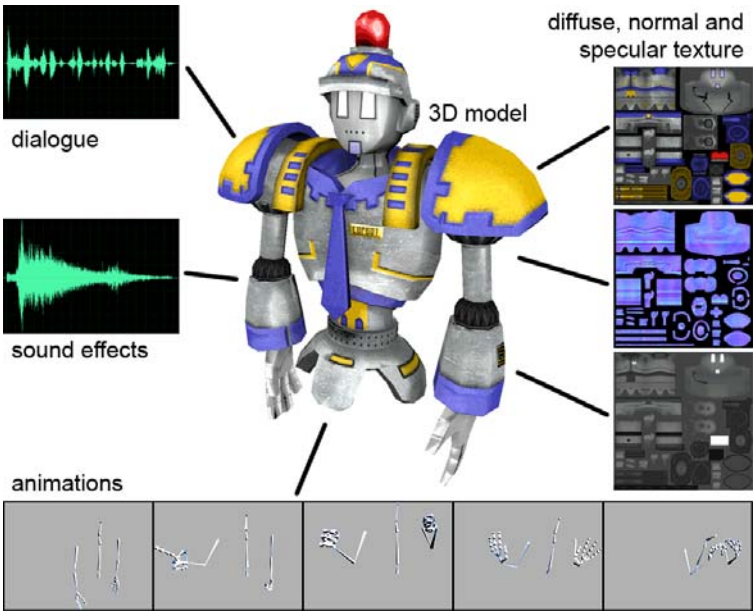


Figure 1a: *A single character already requires many different assets (model © Joost van Dongen).*

Some examples of such assets are a game character model, the texture that is applied to it, the lines of dialogue that the character can utter, and the animations it can perform. A single character might require dozens of animations and lines of text and a game often has dozens of characters. Not only characters, but also levels are needed, maybe weapons, cinematics, music. An average game today easily contains thousands of assets. Although models, textures and levels are usually the most complex assets to handle, the other types of assets require serious attention as well. Sounds need to be converted to the right format, cinematics need to be compressed and scripts may need to be encrypted.

Although code is not considered to be an asset, some ideas from code management are useful for the asset pipeline as well, as will be shown later on in this thesis. In many

ways code is very different from assets, though. The different programming files are much more inter-dependent than other assets and code is much smaller. Even on larger projects, the total size of all the code in the game will probably still be smaller than a single Photoshop file that stores only one texture. Note that unlike C++ code, the small script-files that a level designer might create to, for example, make an enemy move to a certain spot when a door opens, are considered to be assets.

Now that assets are defined, here is Llopis' definition of the game asset pipeline:

"*The content pipeline is the path that all the game assets follow, from conception until they can be loaded in the game*."

The pipeline includes all parts of the path. It includes the usage of content creation tools, exporting the assets from the tools to the game and all the processing performed on them during this path. For example, the general workflow for a graphical artist is to design a model and texture in for instance 3D Studio MAX and Photoshop and then export these into the game. After examining the results in the game, the artist goes back to the content creation tool, tweaks some aspects, and exports again. During exporting lots of things need to be done, usually automatically. Some examples of this processing are converting from one file format to another, repairing broken assets and optimising assets to allow the game to tap into the full power of the hardware.

An aspect that might be overlooked at first, but is equally important, is file management. Files need to be stored somewhere and several people may have to work with them at different times. This requires sharing files among developers, but also locking them to prevent two people from altering the same file at the same time. For the occasion in which someone accidentally removes a file or breaks its contents, the game asset pipeline should include some mechanism to retrieve older versions of these files. Another aspect of file management is automatically generating back-ups for those rare moments when a server crashes.

Now that it is made clear what exactly the game asset pipeline is, lets get back to the claims in the first few lines of this chapter and see what they are based on. Each game shows its assets in a different way. The position of the camera, the lighting, the materials, the effects on the audio, the gameplay: all these things influence how a gamer perceives the assets in the game. For this reason, it is necessary that the asset developer can see how his assets will work out in the game itself. However, the files generated by asset creation tools like Photoshop and 3D Studio MAX are hardly ever suitable to be used directly in a game. This means that if the developer wants to see how his work turns out, he will have to use the exporter to process his work and turn it into something that the game can load. If doing this takes only a second or two, then he will often do this and tweak all the details until they are perfect. However, if it takes him twenty minutes to get his asset into the game and these twenty minutes are filled with the most boring of work, then he will not consider the final tweaks worth the time and might even get frustrated with the waiting time in between.

Even if the developer does not mind to do boring work, it still simply takes time. What takes even more time, is if accidentally an asset is deleted and needs to be remade, or

if two employees work on the same file at the same time and as they cannot merge there changes, one of them has to throw away his work and perform the changes all over again, this time on the result of the work of the other developer. All these things can be prevented by a good asset pipeline and therefore it can really influence the total development time of a game.

How time consuming a bad game asset pipeline can be was shown very clearly by the lawsuit between Activision and Spark [Fleming2007]. One of the mayor points of this lawsuit was that developer Spark had not met several of the deadlines that were set in the contract with publisher Activision. The reason for this was, that Spark had purchased a game engine that was incompatible with many of the features in the asset management system they used. This cost them so much time, that in the end Activision deployed thirty extra developers to help Spark reach its deadlines and sued Spark for the extra costs and the missed deadlines. This exemplifies that a bad game asset pipeline can actually bring a company into legal problems, as missing deadlines can be a reason for a lawsuit or a fine. Although this is an extreme example, it clearly shows that thinking about the game asset pipeline beforehand and putting time into developing it is definitely worth every penny it costs.

## 1.1 The topic of this thesis

The game asset pipeline involves many aspects. From file management and network traffic to mesh optimisation and sound compression and from asset previewing to value tweaking. This thesis will not cover any specific algorithms to perform these tasks, as there are too many of those and most of them are too complex for the scope of this text. There is also no need to discuss them here: every single technique has many articles covering it into the tiniest details and copying these serves no purpose. Instead, this thesis will look into the entire process, the steps it involves and the influences of the game asset pipeline on the design and development process. What things need be taken care of and what optimisations are usually done.

Although the tools for developing the assets themselves, like for example Photoshop, 3D Studio MAX and MAYA, are a part of the game asset pipeline, they will not be discussed here. The developer of the game asset pipeline ‘simply’ has to fit his tools and exporters into these programs, while actually using these programs is solely up to the asset creators. It might be interesting to list the different tools in this thesis and discuss what they are used for and how they might be connected to optimally make use of their features, but again there are simply too many tools for too many purposes to make this feasible and a selection of tools would probably only state the obvious. Also, the asset creation programs are usually chosen by the asset creators, not by the pipeline developer, so the only consideration to the pipeline designer is how to connect the pipeline to these programs.

The main focus of this thesis is twofold. On one side is giving an overview of all the aspects of the game asset pipeline and how they relate to each other. The other focus of this thesis is to analyse the impact that the game asset pipeline has on the design and development process. How does iterative design fit into the pipeline and in what ways are quality and productivity influenced?

This leads to the following research question, which can be split into several sub-questions:

Main question:
*What is the role of the game asset pipeline in the game design and development process?*

Sub-question 1:
*What aspects should be considered when developing a game asset pipeline?*

Sub-question 2:
*How does the game asset pipeline influence the game design and development process?*

Sub-question 3:
*How are the parts of the game asset pipeline related to each other and how do they relate to the rest of the game design and development process?*

## 1.2 The project this thesis is linked to

This thesis is linked to a game development project, which is used as a case study later on. Thesis and project together form what the Utrecht School of the Arts calls the *exegesis*. In the project, known by its working title *IceMagnet*, a team of seven students has chosen to develop a vertical slice of a game of their own design. The author of this text is the project lead and the only full programmer on the team. The rest of the team consists of two game and level designers (Jasper de Koning and Fabian Akker) and four artists (Gijs Hermans, Ralph Rademakers, Martijn Thieme and Olivier Thijssen). Both project and thesis run from March 2007 to the end of August 2007. As the author is also skilled as an artist, developing the game asset pipeline is a task that fits him very well. It requires both knowledge of the technical demands and programming of the game and of the tools that the asset creators use.

IceMagnet is a difficult case to develop a pipeline for. The greatest problem is that the team includes only one programmer, so not all parts of the pipeline can be developed as well as is desirable. This forces the pipeline developer to make the difficult choice of what to develop and what to make the artists do by hand.

## 1.3 Outline of the thesis

This introduction is the first chapter of this thesis. The next chapter analyses all the different aspects of the game asset pipeline in more depth than has been done so far and is based on research into existing literature. It effectively answers sub-question 1. Chapter three answers the second sub-question by looking into the effects that the game asset pipeline has on the game design and development process. It looks into iterative processes, quality, productivity, and (preventing) developer frustration.

The fourth chapter will then show the relations between the aspects of the game asset pipeline through a number of schemes. It also discusses a few examples of the impact of the pipeline on development. All the issues and theory covered in this thesis will then be put to the test in chapter five, which projects this thesis on the IceMagnet project to see how theory works in practice. Chapter six, the final chapter, wraps it all up in the conclusion.

# 2 The components of the pipeline

The pipeline can consist of a large number of different components, which are interconnected in various different ways. Some are quite independent, while others need to work together with virtually everything else. Which parts of the pipeline are used in any given project varies with the demands of that project, as do the exact workings of several parts. This chapter gives an overview of most components and is a summary of [Carter2004], except when otherwise mentioned. It cannot cover everything, however, as each game has very specific requirements.

The figure below shows the relation of the different parts of the asset pipeline, which will be discussed in more depth in the rest of this chapter. The assets are first created using some tool. The tool itself is considered part of the pipeline too, although it is not a large topic from the perspective of the pipeline. The mayor work of the pipeline is the pipeline processing. The assets cannot be loaded directly from tools like MAYA and Photoshop into the game, so they need to be processed first. This processing is the main topic of the game asset pipeline. An important aspect that might not immediately be obvious to those who have not actually been involved in the development of a game, is the content management system. This stores all assets, usually both in the original form and in the exported form. The content management system is of key value to allow developers to work together smoothly in a large team.

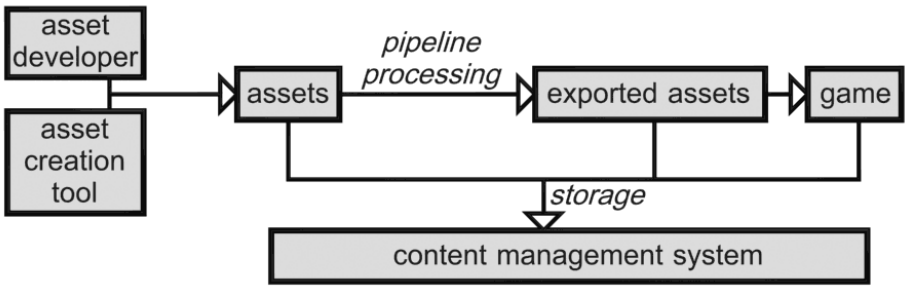


Figure 2a: *A summary of the aspects of the game asset pipeline.*

## 2.1 Content creation

The first part of the asset pipeline has the largest impact on both the production process and the pipeline itself, but is also the least controllable from the perspective of the pipeline developer: the tools with which the design team creates the actual assets. These tools concern the creation and editing of 3D models, textures, animations, music, sound effects, voice recordings, levels and cut-scenes. Most tasks can be performed with existing packages like 3D Studio MAX, MAYA and Adobe Photoshop, while others, especially level design, require tools that are specific for the game or the engine. A level editor can either be developed from scratch or be included with a purchased engine. In some cases an existing tool can be altered or extended to function as a level editor.

Which tools are used is mostly decided independently of the asset pipeline, so the pipeline developer will simply have to cope with what is being used. Most tools allow for

the creation of plug-ins, custom objects and exporters to fit the program with the pipeline. For example, buttons can be added to 3D Studio MAX to automatically export and process the currently opened model and show it in the game engine, thus increasing the simplicity and efficiency of using the exporter. What kinds of plug-ins a tool allows, varies from tool to tool, but usually it is possible to call an external program. This way almost anything can be triggered from the interface of the tool.

## 2.2 Exporting

The exporter is the most well-known part of the asset pipeline. While the file formats used by asset creation tools are made to store as much information as possible for later editing in the tool, the requirements for the file format that is loaded by a game are usually small file sizes and quick loading. This makes the requirements for the files used in the game so different from those that the content creation software uses, that most engines have their own file formats. The result is that to open an asset in the game, it will first need to be exported from the tool to the format used by the game. Usually a plug-in can be created for the tool, so that the user can simply select whether he wants to save the file for use in the game or for storage to allow later editing. The plug-in then automatically performs the exporting. If the tool does not allow the creation of a custom exporter, then some external tool will have to be created to convert from the tool's file format to the game's file format. Note however that the file formats that asset creation tools use are often so complex that it is a very difficult to open them with anything but the asset creation tool itself.

## 2.3 Correctness checks

A task that is often performed during exporting, but can also be placed at other points in the pipeline, is checking the correctness of assets. All games have scores of limitations on assets. In some cases, the tools will automatically enforce these, while in other cases the artist will have to remember the rules and work by them. Some examples of such requirements are that a texture might only be allowed to have a power-of-two resolution (e.g. 64, 128, 256 or 512 pixels for its width and height), or that the name of a file may not contain any spaces (e.g. "BigTree.mesh" is okay, while "Big Tree.mesh" is wrong as it contains a space). Very specific checks can also be performed, like checking whether a character is never taller than three metres. Most of these correctness checks can be performed automatically very easily at some point in the asset pipeline.

## 2.4 Pre-calculations

Some elements needed in the game can be generated automatically. This can be done while loading the level, but in some cases this takes too much time and would have the player wait to play for too long. A very common example of this is calculating the lighting. Many games use detailed, but static lighting, that might take half an hour to calculate. After this calculation phase, the exact lighting of every object in the world is known and can simply be looked up while playing the game. After exporting a level, the lighting can be pre-calculated and stored, so that the game only has to read it from disc, instead of having to calculate it. Automatically pre-calculating such things in the

asset pipeline saves a lot of loading time. Care should be taken, though: some calculations take only a few seconds and can be done during exporting, while calculating the full lighting of a level every time an export is performed takes too long. For such situations, smart optimisations to for example only calculate what has changed are needed. In the most extreme cases, it might be necessary to perform these calculations during the night and work with a less efficient or less good-looking version until the new calculations are finished.

## 2.5 Optimisation

While some pre-calculations are necessary for the game to run, others are only necessary to increase its performance. Performance can be measured in several different ways. The first thing most developers think of with respect to performance is framerate, but it can also mean modifying the assets so that they can be loaded faster, are smaller to store on the disc, or are smaller to store in memory while playing the game. Any of these goals can be the target in the asset pipeline and this can vary among types of assets. An example is storing textures in a compressed format that has slightly less visual quality, but also takes less space to store on disc and in memory and also takes less time to render to the screen. The game asset pipeline can perform tasks like this compression automatically.

## 2.6 Previewing

Most asset creation tools do not show the assets in the exact same way as the actual game does. For this reason, the asset creator will often want to try out his assets in the game to see the result of his work. While newly created levels can simply be loaded in the game to try them, assets like characters are not separately loadable in the game and gameplay might make it difficult to inspect them quickly. Finally, loading the entire game might take half a minute or more, which is a lot of time if it needs to be done often. For this reason, it is often wise to create a simplified version of the game that can load models or other assets and show them to the developer. This previewing version of the game can have an interface to allow close inspection of the asset. Because it only needs to show a specific asset, it does not have to load anything else and can be loaded much faster.

## 2.7 The fast path

The full asset pipeline can take a lot of time to execute. This is a problem if assets need to be previewed or changed a lot and, as will be discussed in the next chapter, might hinder the application of certain development processes. A solution for this is to include a so-called *fast path*. This is a simplified version of the full asset pipeline that takes much less time to execute by leaving out certain parts or doing them less well. This might result in a lower framerate in the game, or a lower graphical quality. However, it is of vital importance to take care that the results do not differ too much, or that the asset creators are aware of the differences. [Carter2004] mentions a case where an artist spends many hours to fix a graphical problem, only to discover that the problem did not exist when using the normal pipeline.

## 2.8 Content management systems

For all of the components of the game asset pipeline that have been discusses so far, it is quite obvious that they are part of the pipeline. Some other aspects might be overlooked when discussing this topic, but are in fact part of it as well. The largest of those is the way files are stored and shared. A very crude way of storing and sharing files, is by creating a shared disc on a server and letting everyone save their files to this disc. This functions, but is extremely prone to problems. If someone accidentally removes a file from the server, or accidentally overwrites the wrong file, then the previous files are lost and will need to be re-created. A more subtle problem is that someone might break an asset during development without anyone noticing it at that time. Weeks later, the problem is noticed, but the older version of the file that did not have the problem is gone now and cannot be retrieved.

Problems like this can be solved by using a more sophisticated content management system to store files. Many existing packages are fit for this job and widely available. Some examples are the open source systems CVS and Subversion (SVN), and the commercial packages Bitkeeper, Perforce and Alienbrain. A discussion of the details of each of these packages can be found in [Carter2004]. In this paper the explanation will be limited to the most important features of content management systems.

Most systems work like a normal file system, allowing users to create directories to order files, while other systems use a database-like system with keywords to find items. The common idea is that each content developer stores a local copy of the files he needs and can work with these as he would do if no special system were involved. Now to use the content management system, there are two actions that the developer can perform. The first is to *commit* changes. This means that any newly created files or changes to files are sent to the central server that keeps track of all the files. Committing is for example done once a day, or when a new asset is finished. The other basic action is *updating*. When updating a file, the developer's computer contacts the server and asks whether any changes have been made to the files. How this works can be seen in the figure below.

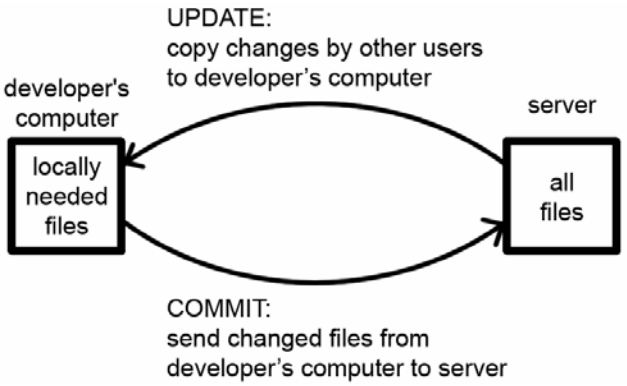


Figure 2b: *How update and commit work with the files on the server and the copies on the developer's computer.*

The benefit of working on the basis of deliberate update and commit commands, is that each individual developer can experiment freely, without running the risk of destroying the version of the assets with which everyone else is working. Once the developer has finished his work and checked whether it is correct, he can then proceed to commit it and thus give others access to his changes or new work.

Each developer's computer simply stores a copy of the files the developer is using, while the server can perform more complicated tasks. One of the most interesting ones is that the server can store older versions of files as well. When a developer commits his work, the old files are not removed. Each file receives a version number from the server and if needed, then the developer can request older versions of files from the server. To allow easier navigation through different versions of files, the developer can add comments to explain what has changed when he commits something.

A few final aspects that are interesting to mention here is that the content management system can be set to automatically mail everyone who is involved with a certain file or group of files as soon as someone commits a change to it. This makes it easy for a lead artist to track what his team is doing. Another feature of the content management system is that the server can be set to automatically create back-ups at set time intervals and that specific user permissions can be set, so that for example an audio designer cannot accidentally remove a texture. Also useful is the ability to lock files, after which no one else can start working on the same file at the same moment.

## 2.9 Meta-data

When developing for several different platforms at the same time, for example for the Nintendo Wii and the Playstation 3, the same assets will be needed in different versions, as each platform has different requirements and possibilities. These files will have to be sorted in some way so that it is easy to get only the files for a certain platform, or to get the corresponding Wii-file for some Playstation-file. Here the content management system comes in handy, as so called *meta-data* can be added to files. This is extra information about each file that the content management system stores to know what the purpose of a file is. Along with what platform the asset is for, the meta-data might also contain information about whether an object is transparent, whether it is a character or a piece of furniture, etcetera. This way all files with a certain property can quickly be found. The down-side of using meta-data is that it only works if the developers consistently fill it in. This is often forgotten or ignored, thus making the meta-data incorrect and useless. Forcing the developer to fill it in when committing a file might be a good idea in this case.

## 2.10 Maintaining the pipeline

Predicting what exactly will be needed from the pipeline at the beginning of a project is as difficult as predicting the exact requirements of the game itself. Changes will surely come during the development of the game and if the pipeline is not created with this in mind, then this can cause some serious headaches and a lot of work to alter things. If for example the format of one type of file changes at some point, then in the worst case

all files of the same type will have to be re-exported. This is a cumbersome task, as each file will have to be manually opened in the asset creation tool to export it.

Several solutions to this problem exist. One might create a tool that converts all files from the old format to the new one, but this is only possible if all necessary information is available in the old files and requires writing an extra tool. Another solution is to add code to the game to be able to load both old and new files. However, if the format changes more often, then this will clutter the code and make it more error prone as many extra things need to be taken into account at loading time. Probably the best solution is to include some kind of intermediate file format for the asset. The exporter saves files into these intermediate file format and then calls another tool to convert the intermediate file to the format that the game actually loads. Now if for some reason the format for the final file changes, then all intermediate files can automatically be converted to the final version. In the intermediate files, efficiency is not an issue, so it is no problem to store a lot of extra data in them that is not needed at the moment, but might come in handy in the future.

An extra benefit of the intermediate files is that they can be in text-format instead of in binary format. This way a developer can easily read or even edit them if necessary, making debugging much more simple. A good choice for the format of the intermediate files is XML. It is widely used and many tools for editing, loading and writing XML are available to quickly work with them.

After updating the pipeline, the changed tools will somehow have to be distributed among all the asset developers. An easy way to force this to happen within one or two days, is by putting the pipeline itself in its entirety on the content management system. If everyone in the team has been conditioned to update every day, then they will automatically receive the updates to the pipeline as well. Content creation tools like 3D Studio MAX can be set to read plug-ins from folders outside the program, so the folders of the content creation tool can be used for the plug-ins. This way the newly updated plug-ins will automatically be loaded by the content creation tool.

## 2.11 Various other pipeline considerations

And endless amount of topics can be discussed in relation to the pipeline. Here some aspects are mentioned that are interesting to think about, but take too much room to thoroughly discuss.

- Bandwidth: if many developers are all committing or updating at the same time, then the server might not be able to handle this much traffic at the same time, thus making all users wait longer.
- Catalogue files: it is often much faster for the game to read large files at once than to read many small files separately. A catalogue file is a file that contains many smaller files. When storage space is limited, it might even be worthwhile to compress the catalogue file using a format like ZIP. The creation of catalogue files can be done automatically by the pipeline.
- Encryption: if security is important, then it might be necessary to encrypt files for the final release of the game. This can also be done automatically by the pipeline.

- Building distribution packages: it is quite common for a publisher to require a playable version to be sent from developer to publisher every few weeks. Extracting the necessary files from the content management system can be automated, although this often requires the proper use of meta-data to know which files are needed for the distribution package.

# 3 Relations to the design and development process

The game asset pipeline is not just a technical problem that needs solving. More or less effort (time and money) can be put into creating the pipeline and this affects the design process. This chapter analyzes what the influences of the pipeline on the design and development process are and shows how the pipeline relates to several types of planning processes.

## 3.1 Planning methodologies and the pipeline

The process of creating a game can be done according to several basic time schemes. The project is planned according to such a system. Game development is a relatively young industry, though, and many companies still lack a proper abstract view on their planning. As the budgets of projects increase, the importance of proper planning and project management increases also and thus more and more game developers are embracing some standard for project management.

### 3.1.1 The Waterfall model

The most straightforward and traditional method is the *Waterfall* model, as is shown in [Hughes2002]. The Waterfall model is also known as *one-shot* or *once-through*. In this model, the project manager plans the entire project from start to finish and it is executed while taking as few steps back as possible. In the case of a computer game, this means that design and development are strictly separated: design is done at the start of the process and development it done after that. All steps in the design and development of the game are planned beforehand.

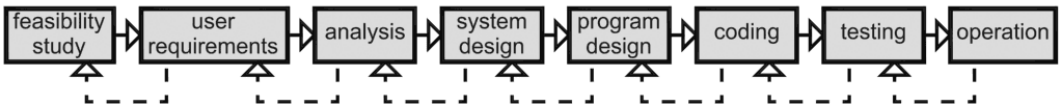


Figure 3a: *A scheme of the steps in the Waterfall model in software development in general [Hughes2002].*

In the case of the Waterfall model, a good asset pipeline is actually not very important. Each asset is expected to be exported and processed only once, or twice in the rare case where something has to be changed. If the pipeline is not run very often, then it is not really a problem if this requires a lot of laborious handwork. A good content management system is much less important as well, because assets are placed there when finished and people rarely need to find and update their old assets. Because the asset pipeline is less important in the Waterfall model, it might be a good idea to spend less time on developing it well. However, the Waterfall model is a very bad idea in games anyway, as is explained below.

### 3.1.2 Agile development

Th Waterfall model only works well if the exact requirements of the project are known beforehand. In games however, many changes are made throughout the project, based on new ideas, user testing, or technical problems that were not predicted beforehand. Having large amounts of changes requires a lot of going back in the process and makes planning the entire project beforehand impossible.

For this reason, game companies often work by iterative techniques. The general name for most iterative techniques is a*gile development* [Highsmith2001]. Whereas traditional design and development processes are based around the goal of eliminating change as much as possible by predicting it and using all kinds of techniques to prevent it, the idea of agile development is to not avoid change, and instead handle it is well as possible. The creation of a product is split into many small parts, which typically take 2 weeks to 2 months to complete. A new part is planned just before it is worked on, instead of planning everything at the beginning of the project. When a part is completed, it is evaluated and changes are made to the planning if necessary. The result is a repeating loop of designing new parts, developing them, testing them, evaluating the results and then re-designing to incorporate the results of the evaluation.

This continuous iteration of designing, creating, evaluating and changing is the basis for agile development and it fits games very well. Whether a game is fun or looks good is very difficult to predict beforehand and new ideas come in all the time during development, so re-evaluating often is a good idea. Some well-known examples of agile methods are Dynamic Systems Development Methodology and Extreme Programming [Hughes2002]. In the field of game development, an often used method is SCRUM [Schwaber1995]. In the SCRUM process, small multidisciplinary teams are formed to work on a single task for 30 days. The members of a SCRUM-team work together very closely. When a 30-day period is finished, the SCRUM are rearranged for the new task, potentially moving half the company around within the building, as the members of a SCRUM-team are supposed to sit close to each other to make communication easy.

Unlike the Waterfall model, agile development and iterative methods in general demand a well designed asset pipeline to function. When working iteratively, assets will be changed often and to evaluate the results, they need to be exported to the game. If exporting takes a lot of time, then this quickly increases the amount of time required to finish work on an asset. The figure below shows how the processing by the pipeline falls into agile development. Iterative design is not limited to periods of several weeks, though. For example, many artists first create a rough design of a character, evaluate its shape, make changes, etcetera. If the changes need to be tested in the game often, then the loop below works on very short periods of only several hours or even less. This puts the pipeline into action very often.

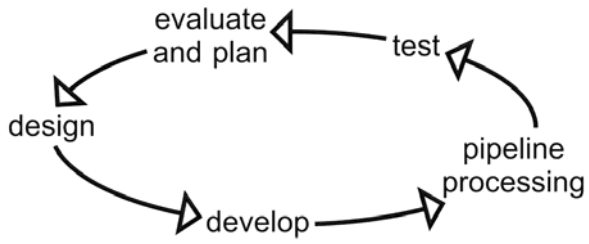


Figure 3b: *The iterative design and development loop, including the game asset pipeline.*

Another aspect of agile development is that less is documented and defined, and that instead there is a constant process of testing the validity of code and assets. It helps if the asset pipeline contains correctness checks that automatically perform a number of tests to see whether an asset is correctly created. This way the pipeline becomes an extension of the automated testing technique that many agile development methods advocate. In automated testing [Beizer1990], a lot of standard operations are automatically performed by the code to see how the game reacts. A character for example might be placed at a certain position, then moved forward one metre and then it is checked whether the position of the character is actually what was expected. When changing the design of the game a lot during the process, it is important to keep testing old assets whether they still work correctly in a changed game and automating this saves time and helps find problems earlier.

When using iterative techniques, new ideas might be tried but removed afterwards as evaluation shows they do not work after all. When using a good content management system, going back to earlier versions is easy to do, whereas it would be impossible if none were used. Many content management systems also allow *branching*. When using branching, a group (for example a SCRUM-team) can work on a separate copy of the game. Updating and committing is still possible, but only have effect on the private branch of the group. When the iteration is finished, the branch is merged with the normal version of the game and only then do the changes made by the SCRUM-team reach the members of the development team that were not in the same SCRUM-team.

The pipeline itself is also part of the agile development method. If changes are made to the design of the game, the requirements of the pipeline might change as well. This turns the pipeline into a system that is constantly under development, undergoing iterative changes often.

All of these aspects make a good game asset pipeline necessary when working with agile development methods.

### 3.1.3 Prototyping

Prototyping is the technique of creating small test-versions of specific aspects of the game to see whether they are fun or functional, or to prove that a technically difficult aspect of the game is possible to do at all. Prototypes are always small and usually intended to be thrown away after creation, keeping only the resulting conclusions and

the lessons learned. Prototyping can be used both in agile development methods and in the Waterfall method. In combination with the Waterfall method, prototyping is used before the main project itself starts off and the conclusions of the prototyping are used to decide whether the project should be done at all and help in planning the aspects that are more difficult to predict. In prototyping, small aspects are designed and then roughly developed to be able to test whether they work.

The most important aspect of prototyping is that is should be fast. The prototype itself remains small and will be thrown away after the results are known, so it is not a problem if things are not made well. Often a technique that will be used on complex assets like characters, might be tested on simple objects like slightly edited boxes that have all the technical aspects of a character, but do not actually look like one. As the asset itself can be made very quickly, it is a relatively greater waste of time if exporting and processing it takes long. For this reason, the asset pipeline is very important for prototyping. Another reason is that with a good content management system, a new idea can easily be tried on the current version of the game and then removed after testing by retrieving an older version of the game assets from the content management system.

## 3.2 Other considerations

So far the relation between the use of certain development methods and the game asset pipeline has been discussed. The game asset pipeline has relations to a number of other aspects of the design and development process as well. These aspects are discussed here.

### 3.2.1 Leading a project

When working with a large team, it can quickly become difficult to know exactly what everyone is doing. The content management system can make it easy for a team-lead to be notified of what his team is doing. Messages can be sent automatically when someone commits new work and marks it as "finished", after which the team-lead can check the quality of the work. Another option if to automatically receive a message when someone has not committed anything that was finished for a while, or did not commit anything at all. It is also possible to check for any developer what his last commits where and thus see quickly what he has been working on recently. If the content management system requires the developers to lock the assets that they are working on, then it even allows the team-lead to see what the members of his team are doing at any specific moment in time. This makes the asset pipeline a useful tool for the team-leads to keep track of what their teams are doing.

### 3.2.2 Teamwork

The asset pipeline can improve teamwork in a number of ways. The first is that properties of assets can be marked and described in the content management system. If someone has to work on another person's assets, then he can quickly see the properties and description of the asset in the metadata that is kept per asset.

An often occurring problem is that two developers edit the same file and then both save their own changes to it. Without a proper asset management system, the result would be that one of the two developers would have to redo his changes on the version that the other developer saved. This wastes time and doing the same work twice is frustrating work. In certain cases, the asset management system can automatically merge the changes made into one file, thus eliminating the requirement to redo changes. This usually only works for text-files though, so this is mainly useful for script-assets and settings. Working on the same file can be totally prevented if the assets can be locked and working on them becomes impossible while someone else has the lock.

A final aspect of teamwork that is helped by the asset pipeline is sending files over to other people. Normally this would be done by sharing them or mailing them or something else, but now the file can simply be committed to the content management system, after which the other developer automatically receives the file on the next update. It is even possible for a texture artist to take a texture, change it, commit the result and the next time anyone updates and plays the game, he will automatically be playing it with the new texture.

### 3.2.3 Quality of work

An asset usually looks different in the game then in the asset creation tool, so working on assets without testing them results in badly adapted assets. If the pipeline is too slow, then asset developers will choose to export as little as possible, thus seeing problems too late and taking extra time to fix them afterwards. The final tweaking of details might even be totally omitted if it takes too much time for the small improvement it yields. This deteriorates the overall quality of the game, as these details often make the difference between a mediocre game and an excellent game.

### 3.2.4 Quality of life

Doing boring work or doing the same work over and over again are probably among the best ways to frustrate and chase away talented developers, especially in a creative business like the games industry, where developers want to be constantly challenged creatively. The parts of development that can be automated by the pipeline are also the most boring parts of it. They are also very error-prone and if something is done wrong, the whole process might have to be repeated. Developing a fast and efficient asset pipeline can help to keep a team motivated and keep developers from applying for a job at the competition.

# 4 The pipeline in schemes and numbers

Literature on the game asset pipeline usually consists of a long list of ideas, issues, best practices and techniques, while little effort is made to create a general, more abstract framework to analyze the pipeline with. This chapter discusses a set of schemes that show the relations between the elements of the pipeline and its users. It also gives some numerical examples of the efficiency of the pipeline.

## 4.1 The pipeline in schemes

### 4.1.1 A broad overview

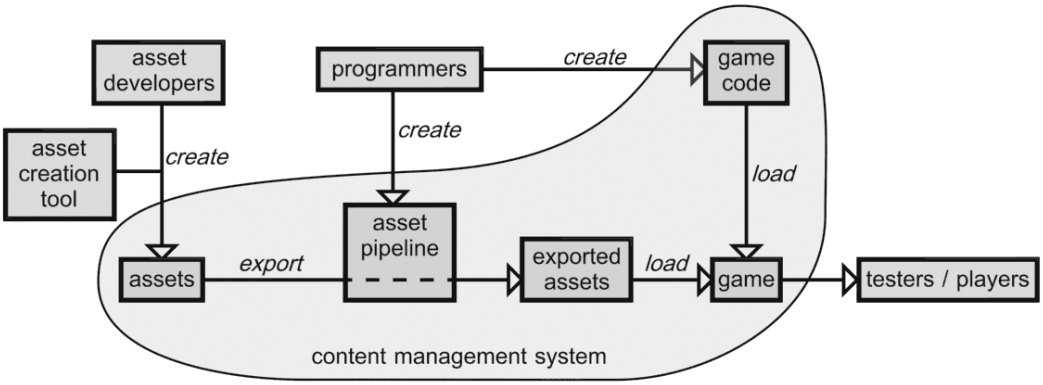


Figure 4a: *A broad overview of the pipeline.*

Figure 4a shows a broad overview of the entire game asset pipeline. The first thing to notice here is how the developers are related to the pipeline. The developers are split in only two groups here: asset developers and programmers. Normally, asset developers are split into further groups, like artists, level designers, game designers, sound designers and composers, or even more sub-groups depending on the size of the team. However, from the perspective of the pipeline, all of these people create assets. They require different specific tools to do their work, but are in essence one group: the creators of assets and users of the pipeline. Programmers are kept apart, because they are the developers of the pipeline and the code they create for the game is not considered to be an asset.

The only part of the game asset pipeline that programmers actually use, is the content management system. This not only contains all the assets and the exported assets, but also the asset pipeline itself, so that updates to the pipeline are automatically sent to all developers. Programmers use the pipeline to manage their code, and also store the latest version of the game itself in the content management system. This way other members of the team can easily try things out in the game and also have the newest proper version of it on their computer.

### 4.1.2 The content management system

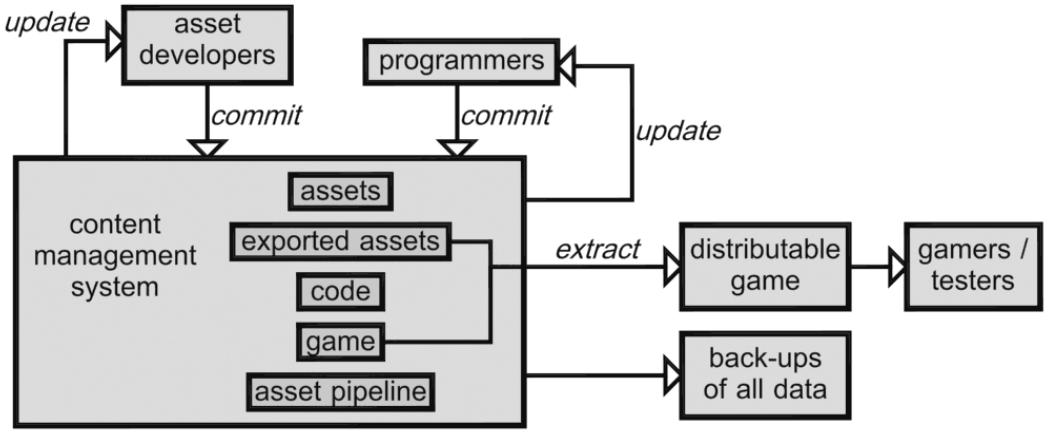


Figure 4b: *The content management system.*

The role of the content management system in the production of a game is shown in figure 4b. The system itself stores the assets, exported assets, code, game and asset pipeline. With the game it means the compiled version of the code. Code is just text and it needs to be compiled to be run by a computer. Artists cannot try things out in the game if they only have the code, so the game needs to be on the content management system. The asset developers and programmers interact with the system by committing their work and updating to get the newest versions of other people's work.

An interesting aspect of the content management system that is shown here, is distributing the game to testers. As testers are often not in-house or at least not linked into the content management system, a distributable version of the game needs to be extracted from the content management system to send to them. This means taking all the exported assets and the game itself and packing it together. The same process is needed to send the latest version to an external producer or client who wants to see the progress, and finally to create the gold version of the game that is released to the marked.

### 4.1.3 The processing in the pipeline

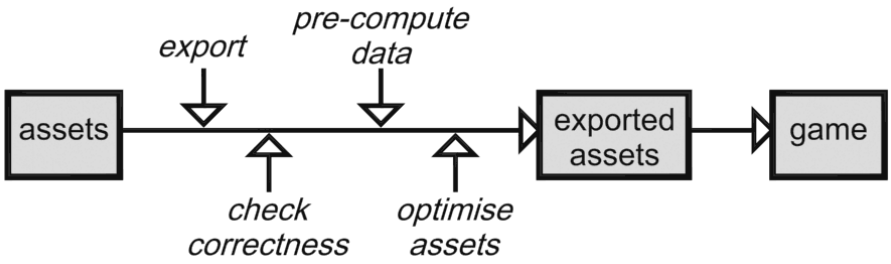


Figure 4c: *The processing steps in the pipeline.*

The final scheme to show here shows the steps in the processing in the pipeline. This scheme shows the four mayor categories of activity and roughly their order in the pipeline. The order is only what is globally done, as several parts of the processing might be interleaved and the order can vary. These are the four categories of processing in the pipeline:

- Exporting is the process of converting data from the asset creation tool (e.g. Photoshop, 3D Studio MAX) to the format that the game can load.
- Checking correctness is the step where bugs in the assets are automatically searched for, like for example audio files that have the wrong bitrate for the game.
- Data that is pre-computed is for example the lighting of an environment or the data needed to plan the paths of the enemies through a level.
- Optimising assets is done to increase the framerate and might for example include cutting objects into pieces to be able to render only the pieces that are visible at a given moment in the game.

Each of these four categories contain many different steps that might be performed there, but most of these are very technical and many are specific for only certain types of games, so there is little use in categorizing them any further here.

## 4.2 The pipeline in numbers

It is impossible to exactly define production time beforehand in games, because there are simply too many factors involved and too many changes occur during the project. However, it is illustrative to have a look at a few theoretical examples of the impact of the game asset pipeline on productivity. Keeping them in mind while deciding how to develop the pipeline helps to make the right decisions.

### 4.2.1 The time spent using the pipeline

It has already been discussed how a good game asset pipeline increases the productivity of asset developers and the quality of the assets. The productivity can be analysed more precisely by calculating the time taken with the exporter. Two examples are given here: an artist doing normal mapping and a level designer working on the positioning of enemies in a level.

The process of creating technical textures like normal maps, specular maps and height maps is difficult for artists, because these are very abstract things. For example, the brightness of a specular map corresponds to how shiny an object is. This correspondence is difficult to grab and predict, because the artist only sees an image with grey-values and needs to predict from this how shiny the object will look in the game. Understanding exactly what a specular map is, is not important for this example. It suffices to know that the artist needs to see how the material of his wall looks in the game again and again to see what is wrong with it and refine its look. Let's say that the artist needs to see his model in the game 10 times and it takes him 8 hours to create the textures. If putting everything in the game takes six minutes, which it might in a very bad asset pipeline, then the total time spent on the pipeline instead of on creating the texture, is 10 x 6 minutes = 1 hour. This increases the time to create these textures

from 8 to 9 hours, costing 12.5% of productivity. If there are nine artists in the team, all facing this same situation, then it is as though eight of them are actually creating assets, while the ninth is doing nothing but working with the asset pipeline. In this case it would be a large win if the pipeline could be made to work faster. It would be even better if the artists could immediately see a preview of their work in the content creation tool, so that they do not need to work with the pipeline to see the result of their work at all. It requires some work from a programmer to make this possible, but this is often time spent well!

The second example is of a designer who is positioning enemies and objects to hide from these enemies. It is difficult to predict exactly what the enemies will do and how fast the player will move, so the designer will want to test the result often. Maybe the wall should be one metre longer, so that the player can reach the next wall exactly in the time it takes an enemy to reload his weapon. In the most common situation in game development, the designer will make changes in the level design tool and then start the game to see the result. 30 Seconds of loading time and 30 seconds of exporting time is not a rare thing in games, and that is excluding the time it might take to walk up to the right spot in the game world. This takes time and is boring for the designer. Now think of the alternative by taking a look at the following example. In the tools of Crytek Studios's game Far Cry [Crytek2004], the designer could press a button at any point in the level editor, and he would be playing the game. The camera used to navigate in the editor simply changes into the game character, the enemies come to life and the designer can try things out. The interaction of the designer with the pipeline itself has been reduced to a single mouse-click and no waiting time at all.

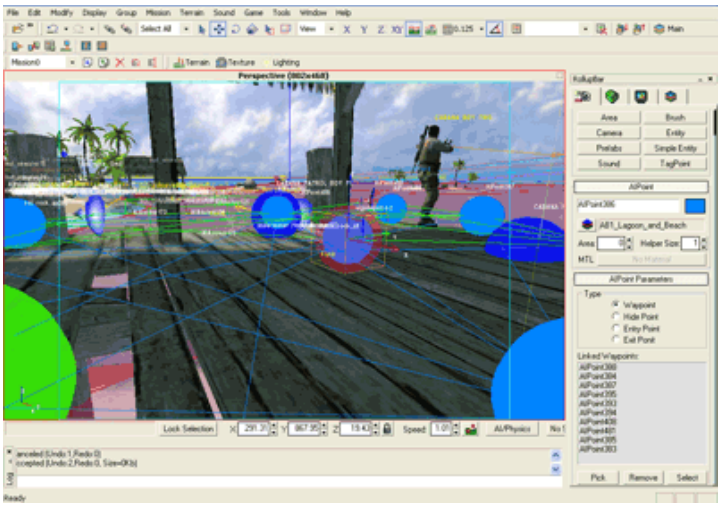

Figure 4d: *A screenshot of the Far Cry level editor.*

### 4.2.2 The cost of creating a good pipeline

A point nicely illustrated by figure 4a, is that there are two groups of people interacting with the asset pipeline. Programmers create it, and asset developers use it to process their assets. The ideal pipeline is one where the asset developers can be as productive as possible, spending as little time with the pipeline itself and getting the best opportunity to work on their assets. However, achieving this ideal requires a lot of work

from the programming team. This results in a balance that must be found in a project. If an improvement of the pipeline saves the artists one minute each time they use it, but the improvement costs a programmer ten hours to develop, then the artists should use it at least 600 times before the costs have been repaid.

An example of how important it is to correctly estimate the amount of work an improvement of the pipeline costs to make and how much work it saves, can be found in the postmortem of Blizzard Entertainment's Diablo 2. Erich Schaefer explains the following:

"*In many cases we created tools to speed up content creation, but then abandoned them. Too often, we decided to keep using inferior tools because we thought we were almost done with the game, and didn't want to take the time to improve them. Unfortunately, in most of these cases, we were not really almost done with the game, and in retrospect a couple of weeks' worth of work would have helped in the year or more of development remaining."* [Schaefer2000].

The important question to ask, is whether the team would be better off with an extra programmer to develop the pipeline, or with an extra artist to actually create content.

# 5 Case study: the IceMagnet project

In the same period in which I worked on this thesis, I also worked on the development of a game with the working title "IceMagnet". This game was developed in over six months by a team of seven students and served as my graduation project for the master in Game Design & Development at the Utrecht School of the Arts. The goal of the IceMagnet-project was to develop a vertical slice of our own, new game concept. This vertical slice was to be used to find a publisher for it and get funding to develop IceMagnet into a full game.

For this project, my role was programming and team management. The rest of the team consisted of four artists and two game and level designers. The scripting language LUA was added to the project, so one of the game designers also programmed the AI and a number of events in this scripting language. As I was the only full programmer in the team, I was also responsible for the development of the asset pipeline. Therefore, this thesis is strongly linked to the IceMagnet project and a lot of ideas that are in this thesis, were tried in a real game development project. This chapter takes a look at the lessons learned during the IceMagnet project.

## 5.1 3D Studio MAX as level editor

We used the graphics engine Ogre 3D [Ogre2007] for our game and although this has many benefits, it misses good tools to work with. This means that I had to develop some way for the level designers and artists to create levels and get them into the game. We used an old version of the tools that I designed for earlier projects and improved on these to be able to create levels. What we ended up with, was that designers created their levels in 3D Studio MAX, had a number of plug-ins for the program to place enemies and the like and an exporter to export their models from 3D Studio MAX to the game. This turned out to be both good and bad.

The benefit of working with 3D Studio MAX is that everyone on the team was already familiar with the package, so there was no learning curve to use the tools. Even more important is that the program already exists, so only the bridge from 3D Studio MAX to the game is needed and the tools to create shapes are already available. As for the things that we needed that were not there, like objects to place enemies: 3D Studio MAX has a scripting language that allows programmers to add anything they want to the package, so it was easy to add extra material properties and the like during the project. Also, 3D Studio MAX is a very diverse program that can do many things and many of these options come in handy during level design. When using a custom level editor instead of 3D Studio MAX, one can never have as many ways to create models as there are in a program like 3D Studio MAX.

This diversity was also the greatest problem with using 3D Studio MAX. 3D Studio MAX has many options that cannot be supported by the game, often because they are simply too complex to do in a real-time game. This means that I had to teach the artists which parts of the program they were not allowed use, because the game did not support them anyway. Also, objects can be broken in many different ways. My exporter

did not support scaling meshes in certain directions, for example, while this is something that artists often do and fixing this afterwards can be problematic.

The main lessen learned here is that a tool should not contain any options that are not supported by the game, because the asset developers need to be taught about which tools they can use and which not and they easily forget these rules, in turn breaking the tools.

## 5.2 A slow exporter

The speed of processing the assets in the asset pipeline is a topic that has been discussed often in this thesis and the IceMagnet project showed exactly why this is important. Because of the slowness of the scripting language in 3D Studio MAX, exporting a full level with a lot of polygons and objects could take up to 15 minutes. This is a long time and because of the way the exporter worked, the computer that was exporting could not be used to do other work in the meanwhile. The result was that designers would often wander around our office, making jokes and keeping other people from their work, because they had to wait for their level to export. A lack of time prevented me from really solving this problem, but a quick solution was developed by allowing designers to mark the objects they had changed and export only these, so that exporting could be done much faster.

## 5.3 Cutting the level into pieces

We were only making one single level for our game, but the amount of effort put into that was enormous, so we needed some way to allow several people to work on the same level at the same time. Our solution was to cut the level into smaller pieces and merge this together to form the entire level in the game. This way, each part of the level could be edited separately by different people at the same time. A nice feature in 3D Studio MAX that helped us here, is to load another file to make it visible, but not editable, so that someone working on one part could see what was going on in another part, without being able to interfere with the work of others.

## 5.4 The lack of correctness checks

Creating animated characters for games is a very difficult task, because the technical limitations that are enforced to keep the framerate high are severe and difficult to understand. I made a list of things that the artists should keep in mind while creating characters, but of course they always missed something, resulting in things that needed to be done again to fix them, costing a lot of time. Although animating characters is always difficult, this problem could have been lightened by creating a small tool that would automatically check a number of rules. Artists could then click this tool once in a while to see whether their mesh was correct with the requirements of the game.

## 5.5 Grass and flowers

One aspect that went very well, was the planting of grass and flowers. We wanted the game to have large amounts of small grass on the ground and the artists should be able to place this grass by hand, so that certain spots could have grass, while others have flowers, or high reed, or just sand and no grass at all. Placing each grass element by hand is a ridiculous amount of work, so I developed a plug-in in 3D Studio MAX with which artists could simply spray grass on the ground by dragging the mouse over it. This works very fast and as it immediately shows the result in 3D Studio MAX, the artists could easily see what they were doing. It did cost a lot of work to develop this tool, though, so it remains uncertain whether this actually saved us any time. It brought us a lot of joy and ease in creating the levels, which is invaluable during crunch time.

## 5.6 The content management system: Subversion

Another smart choice that we made, was to immediately install Subversion [Subversion2007], a free and open source content management system. The great benefit of Subversion is that it implements about all of the aspects that were mentioned in paragraph 2.8 of this thesis as being good to have. Subversion easily allows users to commit changes to the system and get updates from other people's work. It contains versioning, so older versions of files could be retrieved. We had the entire game on the system, so everyone could play the latest version of the game at any moment and if an artist updated a texture, then the level designers would immediately be working with a game that looked better. Because everyone had all the files on his computer, we also had an automatic back-up system that guaranteed that we always had the latest version of everything on seven different computers, reducing the risk of loosing data if a hard-disc would crash.

Subversion also has some downfalls, although these are small in comparison to the benefits we had. The first is that because everyone had all files, updating would occasionally take unnecessarily long. If I switched to a new version of the Ogre 3D engine, then the next day the entire team would have to wait ten minutes to update and receive all the files involved in this, even though they did not need the engine code itself. Another downside of Subversion is that it turns out to be pretty difficult to use for non-programmers, so I had to explain things quite a lot of times. Finally, Subversion does have a form of locking files so that no two people can work on the same file at the same moment, but this locking system does not really work well, so we never used it. Luckily, we did not run into any problems with it either, because we planned well and that means that the same task was never given to two different people. As we were in the same room, any remaining issues were solved by simply asking the rest of the team whether someone was working on a certain file.

## 5.7 Iterative development

Because we were working on an innovative game with a number of gameplay mechanics that have not been used in any game we have seen before, we could impossibly predict exactly which things would work and which would not. This means that we worked extremely iteratively, constantly adding new features and removing or

changing the ones that did not work. Subversion allowed us to easily send new versions of files to each other and update things. The best example of this is how we had a group of text-files for the settings of the game, like the walking speed and the size of the enemies. Because Subversion can merge different versions of text-files, I could program new features and add settings for them while at the same time a game designer was tweaking existing settings. Subversion would automatically detect that we had worked on different parts of the same file and would merge the changes into one new version.

## 5.8 Outsourcing the audio

We did not have any audio-designers in our team, but we had the luck to find a student in sound design and a student in composition who were willing to create our sound and music. Because they were not in the team, we had to find some way to let them develop the audio in the game. I ended up creating a system with which they could add audio files to the game and change sound volumes and variations on their own computers, without the need to come to me to let me program these changes. What they could do with this system was limited, but it gave them enough freedom to do their sound design well. Because we had a separate part of our file system dedicated to the audio, we never hit upon any problems with merging their changes in the audio back into our version of the game.

## 5.9 IceMagnet conclusion

Because I was the only full programmer on the IceMagnet-project and had to program the game itself, I had little time to develop the asset pipeline. For this reason, the knowledge gained in this thesis really helped to make the right choices and see which parts of the pipeline were worthwhile to work on to get the maximum improvement with a minimum amount of time. As has been described in the paragraphs above, a lot of things went wrong, especially with the characters and the use of 3D Studio MAX, but others went remarkably well, like the brush to draw flowers with and Subversion as the content management system. In the end, the rest of the team still spent too much time waiting for the levels to export and the game to load and things like that, so the final result was far from perfect and many lessons have been learned.

# 6 Conclusion

The game asset pipeline is not only an important topic, but also a complex topic that has many different aspects that all need attention. Some of the topics that have been discussed here, are exporting assets from a tool to the game, checking whether they are correctly made, optimising them for better a better framerate and pre-processing them to reduce loading times. The game asset pipeline also helps in making teamwork possible by providing a content management system that can send changes from one asset developer to another and combine their work into one large file system.

The requirements of the pipeline depend on the type of process used during the creation of a game. Especially the iterative development that is often used in game design and development puts a heavy burden on the pipeline, because it requires trying things in the game often. Without a good pipeline, it is doubtful whether real iterative design is a good idea at all, because too much time would be wasted with bad tools, trying to get the work in the game.

This thesis has shown how the game asset pipeline can make or break a game. Not by actually being part of the game itself, but by making the development process either a smooth ride or a drop off a cliff onto sharp rocks. If this thesis has done nothing but convince the reader that the game asset pipeline is a very important part of game design and development that should be thought over and worked on in virtually every project, then this thesis has achieved its goal.

# 7 References


[Beizer1990]
*Boris Beizer*
**Software Testing Techniques**
1990, Van Nostrand Reinhold
ISBN 978-0442206727

[Carter2004]
*Ben Carter*
**The Game Asset Pipeline**
2004, Charles River Media, Inc.
ISBN 1-58450-342-4

[Crytek2004]
*Crytech Studios*
**Far Cry**
2004, Ubisoft

[Fleming2007]
*Jeffrey Fleming*
**Call of Duty: The Lawsuit**
2007, Gamasutra

[Highsmith2001]
*Jim Highsmith and Alistair Cockburn*
**Agile Software Development: The Business of Innovation**
2001, Computer, volume 34, issue 9, September

[Hughes2002]
*Boh Hughes and Mike Cotterell*
**Software Project Management**
2002, McGraw-Hill Publishing Company
ISBN 0-07-709834-X

[Llopis2004]
*Noel Llopis*
**Optimizing the Content Pipeline**
2004, Game Developer Magazine

[Ogre2007]
*The Ogre community*
**Object-oriented Graphics Rendering Engine version 1.4**
2007, www.ogre3d.org

[Schaefer2000]
*Erich Schaefer*
**Postmortem: Blizzard Entertainment's Diablo II**
2000, Gamasutra

[Schwaber1995]
*Ken Schwaber*
**SCRUM Development Process**
1995, OOPSLA'95

[Subversion2007]
*The Subversion community*
**Subversion**
2007, subversion.tigris.org