



VOXEL TERRAIN

Workflow

ABSTRACT

This document aims to explain the workflow used during the development of the game component. It covers the tools that were used, as well as any standards, rules or principles that were followed. The document concludes with a description of the resulting process used to develop new features.

Niklas Johansson

a24nikjo

Introduction

To develop this project, the programming language Java was used together with the low-level library LWJGL. This library contains multiple bindings for libraries typically found in C/C++ programming. These include a binding for GLFW and in turn, OpenGL, which is the API that was used for rendering. This all means that no existing game engine was used, and all such functionality had to be implemented from scratch.

Of course, only the bare minimum of engine functionality was implemented to allow for most of the time to be spent on the game side of things. The use of low-level libraries instead of a game engine does, however, also come with some advantages. The developer has greater freedom when deciding how things should be implemented. Some features could even be impossible, or at least very hard, to implement in mainstream game engines. This is a common problem when developing novel games with complex mechanics.

Low-level programming also results in a better understanding of how things work under the hood. This experience can then be used when dealing with existing game engines. Some design decisions of the game engine might make more sense if you had tried to make something similar yourself. The experience could also make the developer more aware of which operations could be expensive and should be avoided.

Version Control

One thing that I find hard to live without these days is version control. It is an invaluable tool when developing software and was consistently used during the development of both the game and the component. The benefits provided by such a tool not only apply to large teams that want to cooperate but also to smaller teams and solo developers.

One of the main benefits for me as a solo developer is being able to view changes that have been made in chronological order. This provides snapshots of the project at different stages which enables me to go back to see how things worked previously. This can be useful if you for example want to see when a specific bug was introduced or want to roll back an unsatisfactory implementation and start from a clean slate. Version control provides a safety net that allows me to try new things without the fear of breaking anything.

Another great feature of version control is the ability to view a list of changes that have been made since the last commit. This makes the review process a lot easier since I know which files and lines to focus on. It is also possible to get a cumulative list of changes between two different commits. This allows me to view an implementation of a feature consisting of multiple commits as if it were a single commit. This also helps with the review process as I do not have to review every single commit on its own.

The specific version control that was used for this project was Git. The repository was then hosted on GitHub which has the added benefit of acting like a backup storage since the repository exists on more than one computer.

Build Tool

A build tool is used to automate the building process and to separate it from any code editor that you might use. This makes the process of building an application independent of the environment that the developer is using. This is at least the core idea of a build tool, but it can be much more than that. It could handle anything from setting up dependencies to sending an email every time a build fails. The main downside of all this power is the time required to learn the tool and to set up the build scripts. While a build tool might not be necessary for small personal projects like this one, it can still be useful.

The build tool used for this project was Gradle. It is a popular build tool that supports multiple languages but is mainly used for Java and other JVM languages such as Kotlin. This project's main use of Gradle is to handle dependencies and the generation of the project files related to the code editor that was used. While dependencies could have been handled manually as they were very few, it would have required a lot of searching and downloading of libraries. It is much easier to tell Gradle to find libraries of a specific version and set up the project to use them.

It is also good practice to use build tools as the experience gained from using them will become valuable when working on larger public projects with a more complex build process. The process might require the publication of artefacts, tests to pass or the generation of code. It would be unfeasible and error-prone to try to do this manually.

Code Editor

My code editor of choice for working with Java is the Eclipse editor. It is one of the main editors when it comes to Java and comes with a lot of features. The advantage of using a proper editor or IDE as they are called, is to get assistance in developing software. You can still program in a text editor if you like but why make it harder for yourself. The assistance can come in the form of autocomplete, error detection, debugging, runtime configuration and much more. One nice feature that Eclipse has is the ability to format code as well as set up save actions. This becomes very powerful as I can configure a format and then ask the editor to format the code every time I save. This means that most formatting problems disappear automatically. The only downside is that the format configuration is limited to a certain number of options. This could mean that sacrifices must be made when designing your ideal format.

Artificial Intelligence

Another tool that I have started to use a lot is the artificial intelligence application ChatGPT. It serves as both a more powerful search engine and someone that I can share ideas with. Artificial intelligence is a powerful tool, but it is not yet ready to replace a developer fully. This means that anything it says must be verified either by checking sources or asking yourself if the response makes sense. Since I mainly use it to search and discuss ideas instead of asking for full solutions that I copy, these risks are minimized.

As a search engine, I can ask the AI anything from a list of good names for a variable to finding solutions to a problem in the form of algorithms. If the AI can find solutions for the problem, I can then search for them on proper search engines like Google to get more reliable sources.

When discussing ideas, I might present an idea and get constructive criticism. I will of course have an idea of the advantages and disadvantages of the solution beforehand, but the AI can check if I missed something. Another use case is to explain complicated subjects or things with poor documentation. In the case of explaining poorly documented things, one must be careful of inaccurate information. This is due to the AI not having much more information than you and might extrapolate information that is simply wrong.

Coding Standard

This section describes the coding standard that was used. This does not include the formatting that was previously described. This standard is at a higher level than text and can therefore not be formatted automatically in the same way. The advantage of having a code standard is that everyone is on the same page. It would be difficult to navigate the codebase if everyone used their own standards or even worse no standards at all. The downside is the additional time required to apply the standard to all code written.

Member Sorting

The first part of the standard is that members of a class, interface or enum must be in a specific order. There also exist two different orders for classes depending on whether they only contain static members or not. The order itself does not really matter but the fact that all files share this order makes finding members easier. To get an idea of what such an order could look like, the order for a class that contains non-static members is as follows: Static fields, fields, constructors, abstract methods, methods and static methods. Usually, members of the same type in the preceding list are sorted based on their visibility in the order: Public, protected, package private and private. The term usually is used because this can change with the section headers that are described next.

Sections

To make the sorted order of members even more clear, section headers are used. These are big comments with a title describing the type of members that follow it. An example of such a title could be “Static Fields”. To further divide up each section header, sub-section headers are used. These do not indicate to the reader of a new type of member. They are used to group the members of a section into logical groups. For example, the “Methods” section could have the sub-sections “Update” and “Render”. The visibility ordering of the members in a section only applies to each sub-section. This means that the “Update” and “Render” sub-sections could both have members of any visibility with the condition that they are sorted within each sub-section. The benefit of using sections is improved readability as the reader can quickly find what they are looking for and what members are related. The main drawback is the additional time required to sort members and add section header comments.

Variable Prefixes

Another part of the standard that I follow is that all variables with few exceptions must have a prefix. The prefix that the variable receives depends on where the variable comes from. Member variables use “m”, parameter variables use “p” and local variables use “l”. The only exception is for loop integers that currently do not have any prefixes. In the future, these could also be prefixed with the local variable prefix “l”.

The reason for using prefixes is to let the reader know where they can expect to find the declaration or origin of variables. If you read through a method’s content, it becomes very clear which variables are declared within the method, which are declared as parameters and which are declared as members. Like the previous parts of the standard, this results in less time being spent on finding what you are looking for. The only downside is that you must get in the habit of using prefixes. This however is a minor problem and becomes second nature after a while.

Thread Safety Annotations

The last part of the standard is that all methods and constructors must be marked with a thread safety annotation. These annotations do absolutely nothing and act like tags or markers for the reader. They are meant to inform the reader if the method or constructor is thread-safe and if not, which thread may call it. Currently, the following annotations exist: “@ThreadSafe”, “@NotThreadSafe”, “@MainThreadOnly” and “@RenderThreadOnly”.

The “@NotThreadSafe” annotation means that any thread may call the method or constructor, but thread safety is not guaranteed. The reason for having “@MainThreadOnly” and “@RenderThreadOnly” annotations is to warn the user that using any other threads will result in unexpected behaviour. This can happen in certain libraries such as GLFW or OpenGL which expect the main and the render thread to issue all calls. The advantage of these annotations is that a developer can quickly see the intended thread safety of each method. There is no need to inspect the contents of the method or constructor. The disadvantage is again the additional time spent on adding these annotations.

Rules & Principles

Along with the standard come a few rules and principles that do not fit into the concept of a coding standard. These are related more to how the code works and is organized instead of what style it is written in. Such rules and principles depend on the project and to an extent the developers working on it. It can be a good idea to establish these before and during the project to make sure the practices that are deemed bad can be avoided by all involved. Like the code standard, the downside of this is that extra time is spent on making sure the code follows the rules and principles. In this context, the term rule and principle mean almost the same thing and the term rule will be used for both.

Allocation

The first rule is related to allocation and garbage collection. The language used in this project is Java. The language is garbage collected which means that objects are not directly deleted by the programmer. They are instead collected and disposed of by the runtime at a time of its choosing. If the application produces many objects, the garbage collector will trigger more often. When this happens, stuttering can occur which is undesirable.

To prevent this, new allocations should be kept at a minimum. Especially when it comes to code that runs each frame. There is a big difference between allocating something once at startup and once per frame. There are a lot more frames than startups, so the garbage stacks up quickly. The rule is then to avoid allocations in code that runs every frame. This reduces garbage and in turn potential stutters. The drawback here is the added complexity of having to allocate as much as possible outside the core frame loop.

Inner classes

Java supports several constructs to be declared inside each compilation unit. There are classes, interfaces, enums and records. These will hereby be called types. The final rule is that no declaration of types may contain nested types. This ensures that each compilation unit only contains one declaration of a type. The benefit of this is that there are no hidden types and they are all visible as files in the file explorer. There could be cases where the nested type is only used privately by the parent type and no other type should be able to access it. In that case, I would still argue that being able to see exactly what types have been declared without looking through an entire file is better. The only downside is the small amount of additional clutter in the file explorer.

Development

This is the final part of the document which describes the process that was used to develop new features. The description starts with the preparation phase, continues with the execution phase and finishes with the review phase. This is a general description and some variations in the process may occur depending on the feature being implemented.

Preparation

To begin implementing a new feature, some knowledge about the problem and solution is required. I achieve this by first understanding the problem after which I search for a solution. The solution could be an existing algorithm or something designed from scratch. If choosing an existing algorithm, then I study it to become more familiar with how it solves the problem. It would be hard to fully take advantage of an algorithm without a solid understanding of how it works.

Once I know what the solution should be, the actual implementation can begin. The first step is to add any new dependencies that I require for the implementation. This is done in the Gradle build script and only requires the name and version of the library. A refresh of the Eclipse project files is also conducted via the Gradle. This will include the new dependencies in the Eclipse project.

The final part of the preparation is to implement any functionality in existing systems that is required by the solution. A texture class might for example be required if the solution uses textures. This texture class would handle any interaction with low-level libraries and would enable the implementation of the solution. Another example could be if your vertex data initially only contained floats, but the solution requires integer vertex attributes. This would mean that support for integer vertex attributes must be added before the solution can be implemented. After I have finished adding all the required functionality, the implementation of the solution can begin.

Execution

This is the phase in which the proposed solution gets implemented. The phase is quite straightforward as I already know how to go about implementing the solution from the preparation that was conducted in the previous phase. I can always consult ChatGPT if I get stuck on something or if I cannot figure out why something is not working. If the solution consists of logical parts, then I like to commit several times during the phase. This makes it easier to understand the history and is preferred to a massive single commit. When the implementation is done, the next phase begins.

Review

In this final phase, the quality and functionality of the changes made are reviewed. The functionality of each change is inspected to detect any bugs or potential problems with the change. This could include mismatches between the expectations of a caller of a method and the method's implementation. This is sort of like design by contract but since the language offers little support for this, I must do it in my head during this phase. Each change is also checked against the format, code standard and rules described in this document. The formatting is done automatically but the checking of the others requires manual inspection. This phase is basically a code review where I review my own code. As previously stated, this phase is made easier by having Git provide a nice list of what changes have been made. Without it, I would have to remember modifications that were made during the entire implementation which would not be feasible.