TGDocument

MrTroble Clazilla IceDynamix

Contents

1 Introduction and Goals

1.1 Requirements Overview

The goal is to create a small game, representing the functionality of the engine. It should run reasonably well. Any kind of underlying game design, such as level design, UX and such are not to be considered at the current stage, as our only focus is the underlying engine.

1.2 Quality Goals

1. Good performance of engine
   - Example: High framerate, responsive UI
2. Clean codebase with documentation
   - Example: Easy to understand code, with documentation viewable on a website
3. Stability of the engine
   - Example: No crashes upon various actions with the engine window

1.3 Stakeholders

- MrTroble, Discord: MrTroble#5516
  - Makes the most decisions regarding the underlying architecture and needs to be able to implement features to all parts of the architecture
- IceDynamix, Discord: IceDynamix#7153
  - Implement all automated tasks regarding the project, implement features for the ShaderTool, and implement parts of the editor
- Clazilla, Discord: Clara#5179
  - Design the resources for the game and provide features for the ShaderTool

2 Architecture Constraints

The whole architecture of all our projects is generally constrained to cache friendly and low overhead operations.

2.1 Disallowed

Therefore, following things are not allowed.

Disallowed	Description	Example
Exceptions	We don't want our game to crash midway. This adds a lot of unnecessary overhead (5)	
Cache unfriendly Container	A big risk to miss the L1 cache, which is the fastest cache (3)	`std::map, std::list, ...`
Per object function	On a large scale those functions pile up a lot of overhead (5)	`texture.create();`
Object-Orientation	The basic nature of object orientation misses the point of data transformation (5)	`texture.getWidth(); // No optimization guarantee`
DLLs	They remove the the optimizers ability to optimize, which is really bad (2)	
runtime polymorphism	This hits cold memory and misses the branch prediction	`virtual toString();`

2.2 Reduced allowed

We try to reduce the usage of following things in performance critical systems.

Reduced allowed	Description	Example
Template libraries	While template libraries offer a great amount of flexibility and freedom, they also come at the cost of overhead, as the typesystem has to figure out the input type at runtime (4)	`glm::translate`
Functions in structs or classes	This should only be used by entries on non-critical paths, as it possesses a big risk of potential overhead (4)(5)	`struct _Test { const char* getTestName(); };`
Global variables	This adds startup overhead, so better not have too much (1)	`extern int x = 0;`

Reduced allowed	Description	Example
Copy and Swap	Try to avoid unnecessary copies and swaps	`std::string name = "Test"; std::string name2 = name; //Copy`
High level abstraction	Every abstraction comes with a cost, the lower the level the better (4)	Code Generation, Classes, Templates
Smart pointer	They have hidden costs and threading issues (4)	`std::unique_ptr<int>`

2.3 Encouraged

In contrast to the list above, it is strongly recommended to use the following patterns and techniques.

Encouraged	Description	Example
Cache friendly containers	Containers that barely miss L1 cache	`std::vector`, `std::array`
`new` allocations	Dynamic memory allocation (malloc)	`char* chars = new char[x]`
low level abstractions	This reduces abstraction cost	such as functions
Structs	No need to worry about visibility	`struct T { int x; }`
Namespaces	Every code should be within a namespace to reduce ambiguity	`namespace tge::test {}`
Macros	Macros can shift some performance cost to compile time	`#define CHECK(x) if(x) {}`
`std::atomic`, `VkFence` ...	For thread safety	`std::atomic<bool>`
Fixed memory allocation	Reduces the cost of dynamic allocation	`char test[25]`
`inline`	Encourages the compiler to inline the function, to reduce calling overhead	`inline void test() {}`
`noexcept`	To be extra sure there are no exceptions	`void test() noexcept {}`

Encouraged	Description	Example
Error return codes	If there's the need for error handling	`if(vkCreateDevice(...))`
Small size optimization	Use pointers in dynamic lists and allocate the contents differently, when they are bigger	`std::vector<Test*>`
`constexpr`, `consteval`	This moves cost from the runtime to the compile time	`constexpr uint32_t test = 32`
`const`	Gives the compiler a better base to optimize	`void test(const char* name);`
GPU Memory	Everything should be copied to GPU memory as soon as possible	
`vector::reserve`	Use `reserve` or `resize`, before using a vector to reduce the relocations	`vec.reserve(200);`

2.4 Additional

The systems need to run on different hardware whom themselves may have additional hardware restrictions those should always we queried and cached while starting up. Furthermore because of the Vulkan API, which the Engine and therefore a large part of our systems are based on, enforces a lot of other restrictions, such as GPU memory offsets, whom can also differ between hardware. Refer to The Vulkan Specification for more information. The project is currently required to use MSVC 2019 or newer as the compiler. O2 optimization is being used in release mode. The software requires any sort of graphics module, which supports the Vulkan API, as hardware. This can either be a onboard graphics chip or a full-on card. This also should run on all x86 and x64 processors.

We also have a set of style guidelines for contributions to our repositories. Refer to troblecodings.com

2.5 Sources

(1) CppCon 2018: Matt Godbolt "The Bits Between the Bits: How We Get to main"

(2) CppCon 2017: James McNellis "Everything You Ever Wanted to Know about DLLs"

(3) CppCon 2014: Chandler Carruth "Efficiency with Algorithms, Performance with Data Structures"

(4) CppCon 2019: Chandler Carruth "There Are No Zero-cost Abstractions"

(5) CppCon 2014: Mike Acton "Data-Oriented Design and C++"

3 System Scope and Context

The Engine is based upon the Vulkan API. The Vulkan API's primary purpose is to provide platform and vendor independent GPU driver access. The engines shader programs, as well as every call to the GPU, go through the Vulkan API and the driver. Any game build by the engine, as well as the editor, is built upon the engine systems. Our tool system (ShaderTool) will be accessed through the Editor UI or per command line by an other programmer.

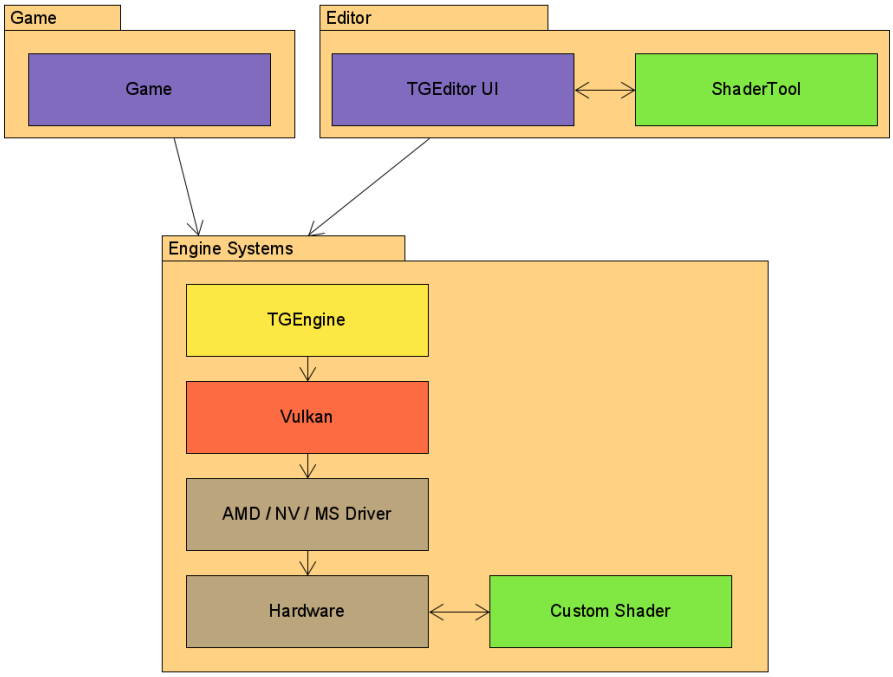

Figure 1: System Scope

4 Solution Strategy

Contents

A short summary and explanation of the fundamental decisions and solution strategies, that shape the system's architecture. These include

- technology decisions
- decisions about the top-level decomposition of the system, e.g. usage of an architectural pattern or design pattern
- decisions on how to achieve key quality goals
- relevant organizational decisions, e.g. selecting a development process or delegating certain tasks to third parties.

Motivation

These decisions form the cornerstones for your architecture. They are the basis for many other detailed decisions or implementation rules.

Form

Keep the explanation of these key decisions short.

Motivate what you have decided and why you decided that way, based upon your problem statement, the quality goals and key constraints. Refer to details in the following sections.

5 Building Block View

5.1 ShaderTool

The primary purpose of the ShaderTool was to compile shader into the Engine. The tool started to grow because of its console interface, which makes it ideal for tooling.

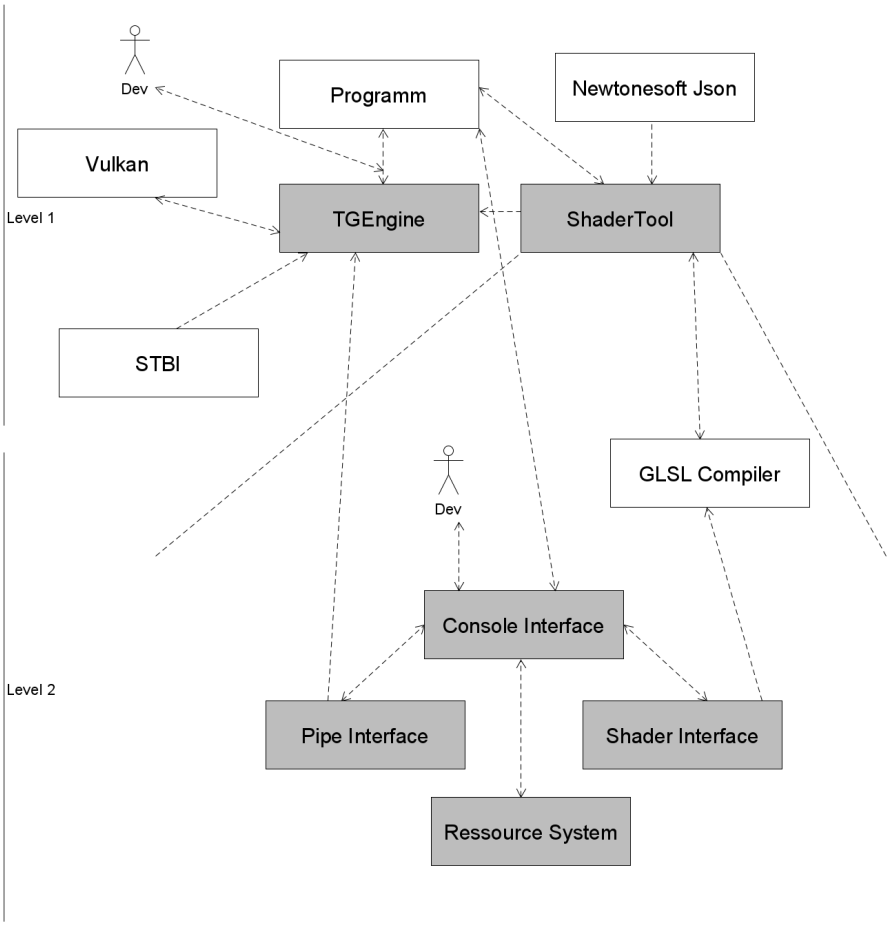

Figure 2: Whitebox

Subsystem	Description
Console Interface	Allow s for console input and forwards the arguments to the given subsystem
Shader Interface	This system compiles shader and embeds them into the engine
Pipe Interface	Analyzes the compiled shaders and creates the according pipeline information
Resource System	This system administrates all game resources.

5.2 TGEngine

The decomposition has historical reasons

Subsystem	Description
IO	For general in and output managment
UI	Contains all user interface components
Gamecontent	Everything that can be put into the game (Actors, Lights ...)
Pipeline	This is the heart of the engine and contains everything needed to create a Vulkan Application with the according featureset needed
General IO	Platform independency layer for file handling
Properties	A small XML reader with a given schema
Resourcesystem	the counterpart to the ShaderTools resource system, except this one only reads the resources
Memory Layer	The own memory management for the GPU, as well as the system memory
Pipeline Stages	Consists of all passes and commands recorded to the GPU
Buffer	All the static memory buffering on the GPU

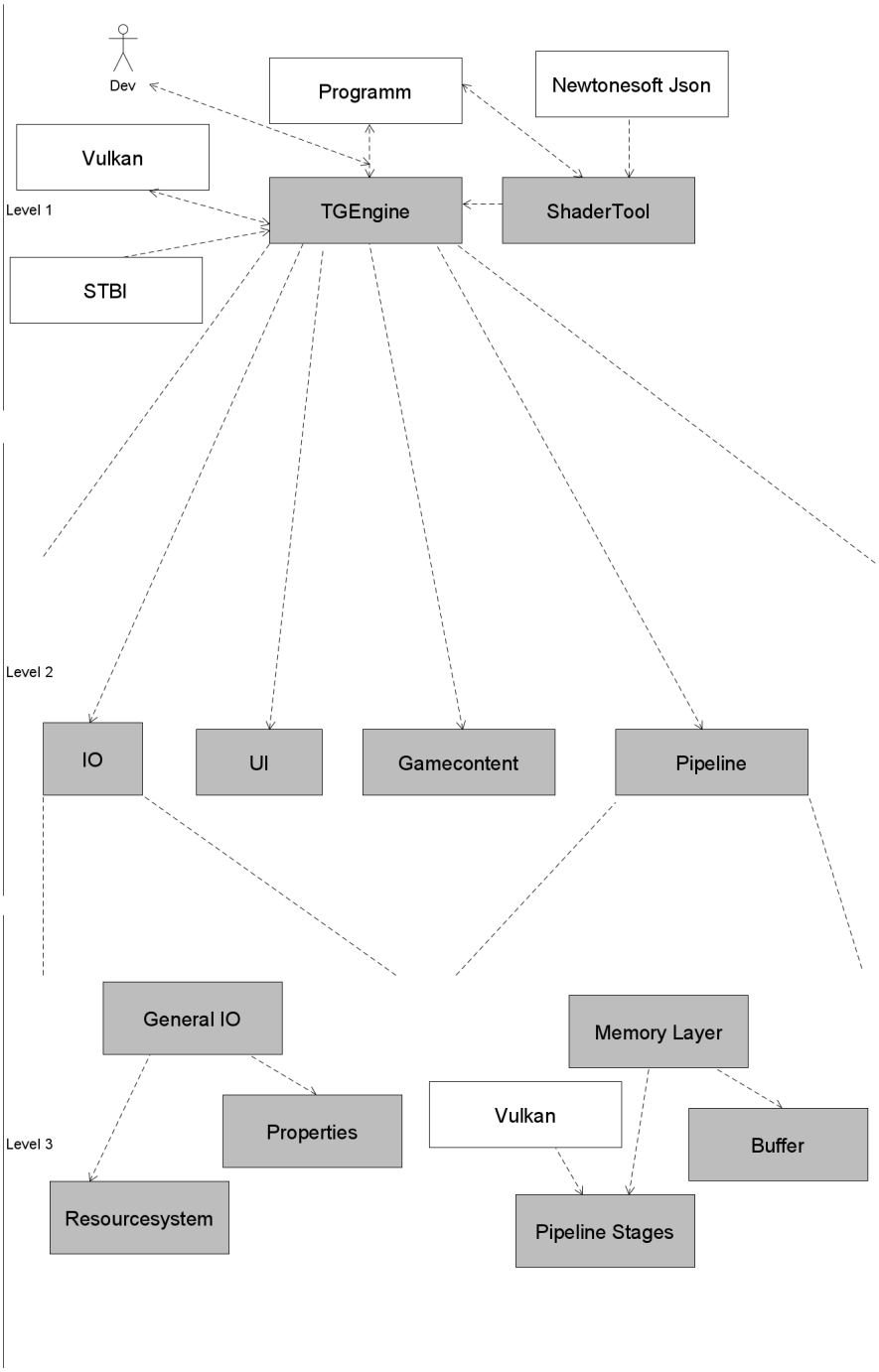

Figure 3: Whitebox

6 Runtime View

Contents

The runtime view describes concrete behavior and interactions of the system's building blocks in form of scenarios from the following areas:

- important use cases or features: how do building blocks execute them?
- interactions at critical external interfaces: how do building blocks cooperate with users and neighboring systems?
- operation and administration: launch, start-up, stop
- error and exception scenarios

Remark: The main criterion for the choice of possible scenarios (sequences, workflows) is their **architectural relevance**. It is **not** important to describe a large number of scenarios. You should rather document a representative selection.

Motivation

You should understand how (instances of) building blocks of your system perform their job and communicate at runtime. You will mainly capture scenarios in your documentation to communicate your architecture to stakeholders that are less willing or able to read and understand the static models (building block view, deployment view).

Form

There are many notations for describing scenarios, e.g.

- numbered list of steps (in natural language)
- activity diagrams or flow charts
- sequence diagrams
- BPMN or EPCs (event process chains)
- state machines
- ...

6.1 <Runtime Scenario 1

- *<insert runtime diagram or textual description of the scenario>*
- *<insert description of the notable aspects of the interactions between the building block instances depicted in this diagram.>*

6.2 <Runtime Scenario 2

6.3 <Runtime Scenario n

7 Deployment View

The following chapter describes the different build environments and the artifacts that each one produces. This also covers how and for what they are used.

To build your own versions you need to download all dependencies using the setup.py script in the engine repository (or submodule). Otherwise it will not compile. For more information see https://troblecodings.com/

7.1 TGEngine and ShaderTool

This part is about the TGEngine repository. This repository should not be used by the engine users. For user who want to make a game with the engine you should use the Template repository or use that repository as submodule.

Like Google, we want to enable the compiler to optimise our code hence we use static linkage.

There is no auto deployment for anything but the ShaderTool. The artifacts produced by the compile pipeline can be accessed through the artifacts tab on the given build. This produces a runnable dotnet core application. This should be able to run through the dotnet command on Linux and Mac. On windows you just need to execute the given executable (.exe) file. The ShaderTool manages your projects and resources. The engine itself produces a static library, which can be used to link your project against. The engine itself currently only works on Windows systems with the according Vulkan 1.0 compatible graphics device. For more information on whether your system is Vulkan capable or not please visit the gpuinfo database. *Note: There is a working Linux compile chain, however there is currently no demo as the window creation is still missing*

7.2 Template

This part is about the Template repository. This repository should normally be used to create a new game project. However as there is currently a GitHub bug that prevents the submodule template to work correctly. Hence we recommend to manually install the submodule and still use the project, but reset the contents. A getting started page is currently being worked on. For a example on how that could look see TGTest. This system should produce a runnable with the same restrictions as the engine itself.

7.3 TGTest

This part is about the TGTest repository. This is a test repository to show off current features and general usage of the engine. This again produces a runnable file with the same restrictions as the engine itself. This already contains a version of the Engine as submodule.

7.4 TGEditor

The TGEditor repository holds the editor source code and it's resources. The editor is built upon the engine and is used as GUI for the ShaderTool. Therefore it produces a runnable file as well as a TGEngine Resource File (.tgr) file containing the baked resources from our resource system. On top of the restrictions applied by the engine the editor needs a working ShaderTool artifact, which is included in the repository under the TGEditor folder.

7.5 TGDocument

This is the repository which this documentation is saved in. The HTML Version is available here: TGDocument

8 Glossary

Contents

The most important domain and technical terms that your stakeholders use when discussing the system.

You can also see the glossary as source for translations if you work in multi-language teams.

Motivation

You should clearly define your terms, so that all stakeholders

- have an identical understanding of these terms
- do not use synonyms and homonyms

Form

A table with columns <Term> and <Definition>.

Potentially more columns in case you need translations.

Term	Definition
<Term-1>	*<Definition-1>*
<Term-2>	*<Definition-2>*