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

Contents

System scope and context - as the name suggests - delimits your system (i.e. your scope) from all its communication partners (neighboring systems and users, i.e. the context of your system). It thereby specifies the external interfaces.

If necessary, differentiate the business context (domain specific inputs and outputs) from the technical context (channels, protocols, hardware).

Motivation

The domain interfaces and technical interfaces to communication partners are among your system's most critical aspects. Make sure that you completely understand them.

Form

Various options:

- Context diagrams
- Lists of communication partners and their interfaces.

3.1 Business Context

Contents

Specification of **all** communication partners (users, IT-systems, ...) with explanations of domain specific inputs and outputs or interfaces. Optionally you can add domain specific formats or communication protocols.

Motivation

All stakeholders should understand which data are exchanged with the environment of the system.

Form

All kinds of diagrams that show the system as a black box and specify the domain interfaces to communication partners.

Alternatively (or additionally) you can use a table. The title of the table is the name of your system, the three columns contain the name of the communication partner, the inputs, and the outputs.

<Diagram or Table>

<optionally: Explanation of external domain interfaces>

3.2 Technical Context

Contents

Technical interfaces (channels and transmission media) linking your system to its environment. In addition a mapping of domain specific input/output to the channels, i.e. an explanation with I/O uses which channel.

Motivation

Many stakeholders make architectural decision based on the technical interfaces between the system and its context. Especially infrastructure or hardware designers decide these technical interfaces.

Form

E.g. UML deployment diagram describing channels to neighboring systems, together with a mapping table showing the relationships between channels and input/output.

<Diagram or Table>

<optionally: Explanation of technical interfaces>

<Mapping Input/Output to Channels>

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 it's console interface which makes it ideal for tooling.

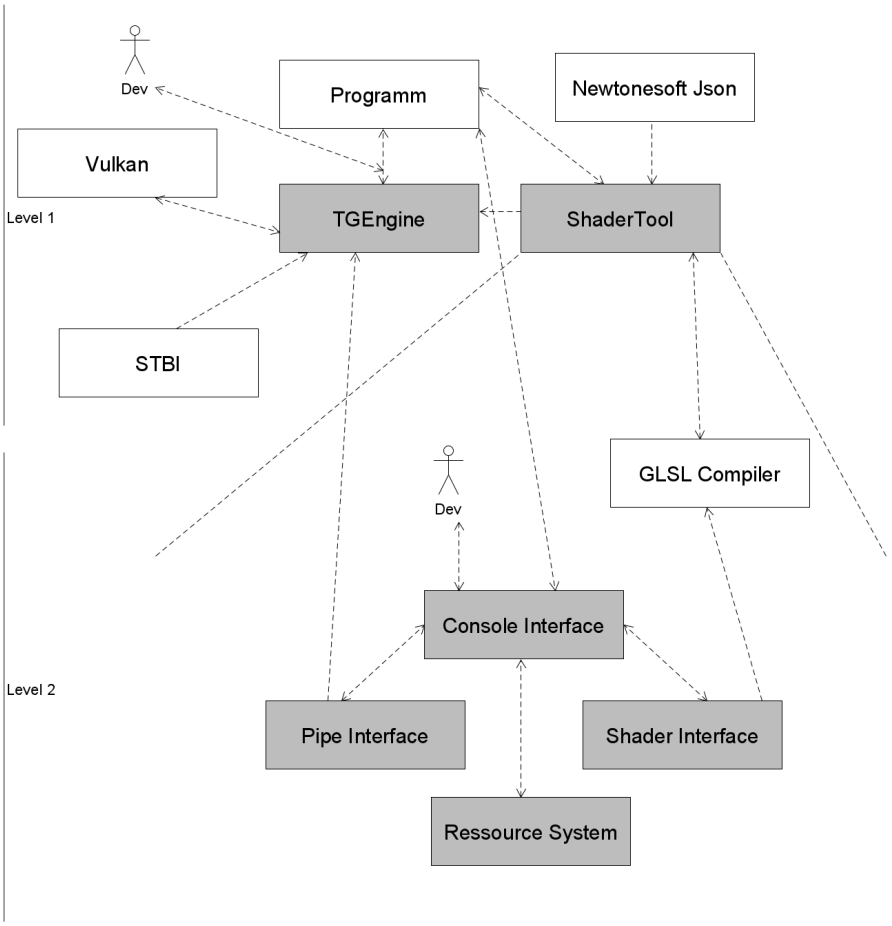

Figure 1: Whitebox

Subsystem	Description
Console Interface	Allowes for Console input and forwards the arguments to the give 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
Ressource System	This system administrates all game ressources.

5.2 TGEngine

The decomposition has historical resons

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
Ressourcesystem	the counterpart to the ShaderTools Ressource System, except this one only reads the ressources
Memory Layer	The own Memory managmet 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

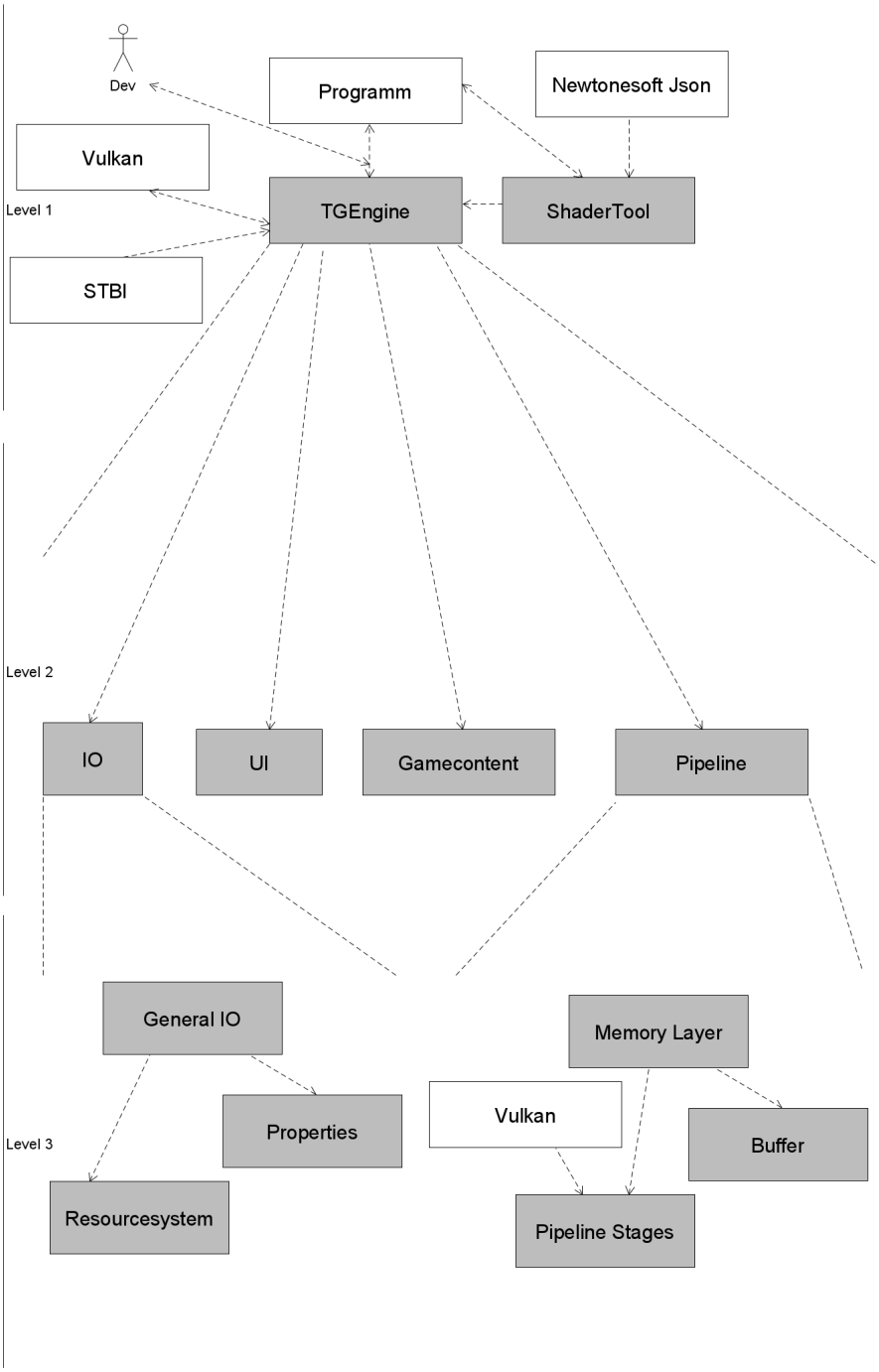

Figure 2: Whitebox

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

Content

The deployment view describes:

1. the technical infrastructure used to execute your system, with infrastructure elements like geographical locations, environments, computers, processors, channels and net topologies as well as other infrastructure elements and
2. the mapping of (software) building blocks to that infrastructure elements.

Often systems are executed in different environments, e.g. development environment, test environment, production environment. In such cases you should document all relevant environments.

Especially document the deployment view when your software is executed as distributed system with more then one computer, processor, server or container or when you design and construct your own hardware processors and chips.

From a software perspective it is sufficient to capture those elements of the infrastructure that are needed to show the deployment of your building blocks. Hardware architects can go beyond that and describe the infrastructure to any level of detail they need to capture.

Motivation

Software does not run without hardware. This underlying infrastructure can and will influence your system and/or some cross-cutting concepts. Therefore, you need to know the infrastructure.

Maybe the highest level deployment diagram is already contained in section 3.2. as technical context with your own infrastructure as ONE black box. In this section you will zoom into this black box using additional deployment diagrams:

- UML offers deployment diagrams to express that view. Use it, probably with nested diagrams, when your infrastructure is more complex.
- When your (hardware) stakeholders prefer other kinds of diagrams rather than the deployment diagram, let them use any kind that is able to show nodes and channels of the infrastructure.

7.1 Infrastructure Level 1

Describe (usually in a combination of diagrams, tables, and text):

- the distribution of your system to multiple locations, environments, computers, processors, .. as well as the physical connections between them
- important justification or motivation for this deployment structure
- Quality and/or performance features of the infrastructure
- the mapping of software artifacts to elements of the infrastructure

For multiple environments or alternative deployments please copy that section of arc42 for all relevant environments.

<Overview Diagram>

Motivation *<explanation in text form>*

Quality and/or Performance Features *<explanation in text form>*

Mapping of Building Blocks to Infrastructure *<description of the mapping>*

7.2 Infrastructure Level 2

Here you can include the internal structure of (some) infrastructure elements from level 1.

Please copy the structure from level 1 for each selected element.

7.2.1 *<Infrastructure Element 1

<diagram + explanation>

7.2.2 *<Infrastructure Element 2

<diagram + explanation>

...

7.2.3 *<Infrastructure Element n

<diagram + explanation>

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>*