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

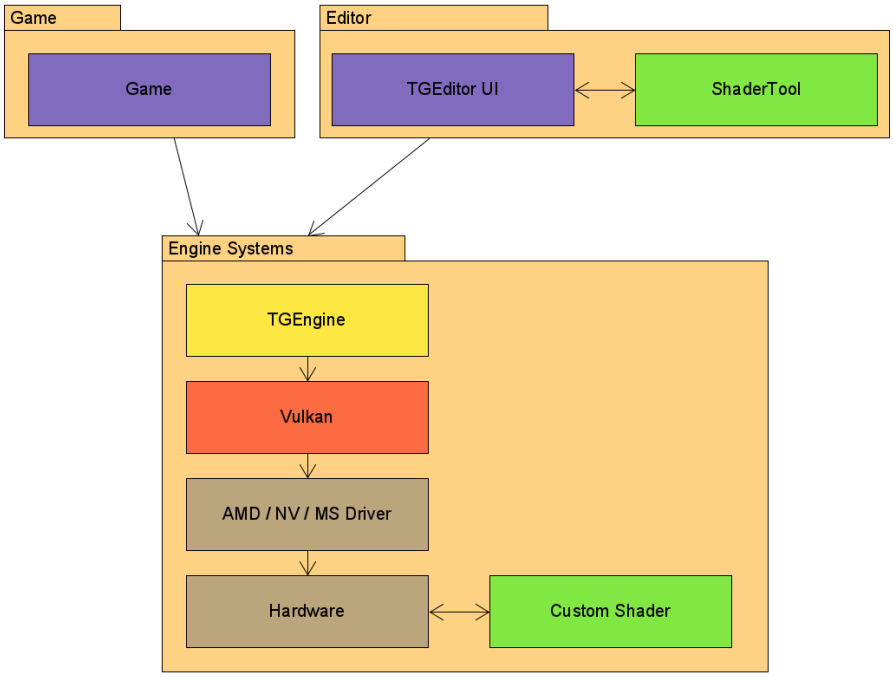

Figure 1: System Scope

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

Content

The building block view shows the static decomposition of the system into building blocks (modules, components, subsystems, classes, interfaces, packages, libraries, frameworks, layers, partitions, tiers, functions, macros, operations, datas structures, ...) as well as their dependencies (relationships, associations, ...)

This view is mandatory for every architecture documentation. In analogy to a house this is the *floor plan.*

Motivation

Maintain an overview of your source code by making its structure understandable through abstraction.

This allows you to communicate with your stakeholder on an abstract level without disclosing implementation details.

Form

The building block view is a hierarchical collection of black boxes and white boxes (see figure below) and their descriptions.

Level 1 is the white box description of the overall system together with black box descriptions of all contained building blocks.

Level 2 zooms into some building blocks of level 1. Thus it contains the white box description of selected building blocks of level 1, together with black box descriptions of their internal building blocks.

Level 3 zooms into selected building blocks of level 2, and so on.

5.1 Whitebox Overall System

Here you describe the decomposition of the overall system using the following white box template. It contains

- an overview diagram
- a motivation for the decomposition
- black box descriptions of the contained building blocks. For these we offer you alternatives:
  - use *one* table for a short and pragmatic overview of all contained building blocks and their interfaces
  - use a list of black box descriptions of the building blocks according to the black box template (see below). Depending on your choice of tool this list could be sub-chapters (in text files), sub-pages (in a Wiki) or nested elements (in a modeling tool).

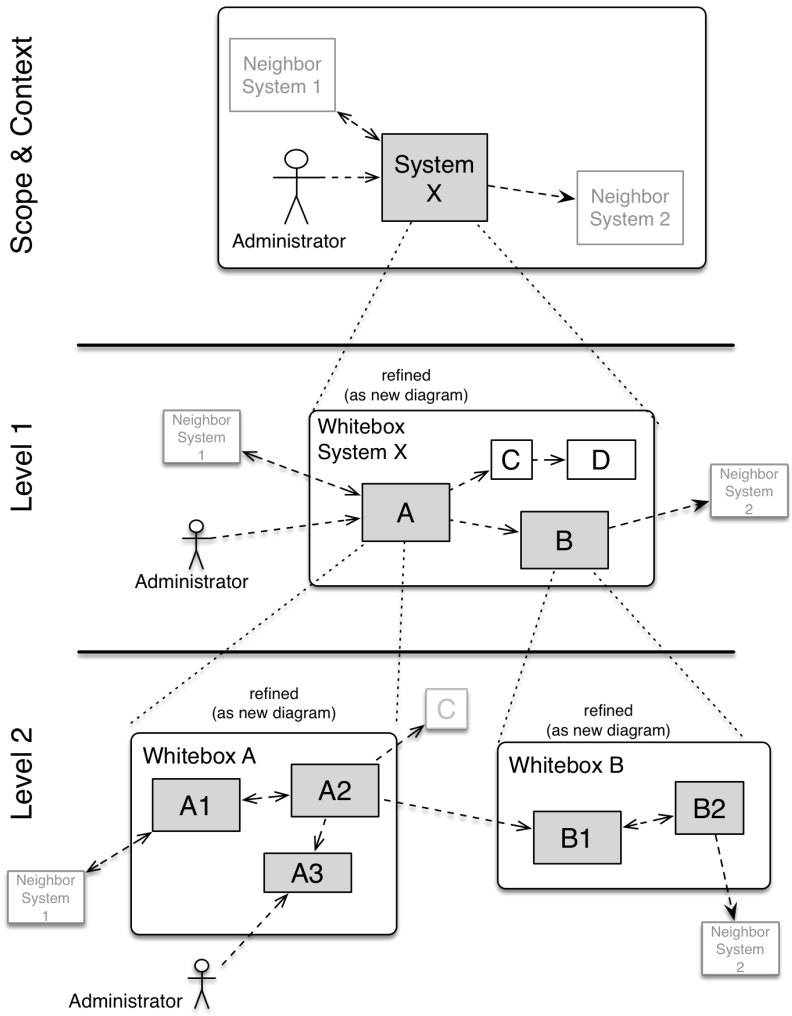

Figure 2: Hierarchy of building blocks

- (optional:) important interfaces, that are not explained in the black box templates of a building block, but are very important for understanding the white box. Since there are so many ways to specify interfaces why do not provide a specific template for them. In the worst case you have to specify and describe syntax, semantics, protocols, error handling, restrictions, versions, qualities, necessary compatibilities and many things more. In the best case you will get away with examples or simple signatures.

<Overview Diagram>

Motivation *<text explanation>*

Contained Building Blocks *<Description of contained building block (black boxes)>*

Important Interfaces *<Description of important interfaces>*

Insert your explanations of black boxes from level 1:

If you use tabular form you will only describe your black boxes with name and responsibility according to the following schema:

Name	Responsibility
<Blackbox 1>	*<Text>*
<Blackbox 2>	*<Text>*

If you use a list of black box descriptions then you fill in a separate black box template for every important building block . Its headline is the name of the black box.

5.1.1 <Name black box 1

Here you describe <black box 1> according the the following black box template:

- Purpose/Responsibility
- Interface(s), when they are not extracted as separate paragraphs. This interfaces may include qualities and performance characteristics.
- (Optional) Quality-/Performance characteristics of the black box, e.g.availability, run time behavior, ...
- (Optional) directory/file location
- (Optional) Fulfilled requirements (if you need traceability to requirements).
- (Optional) Open issues/problems/risks

<Purpose/Responsibility>

<Interface(s)>

<(Optional) Quality/Performance Characteristics>

<(Optional) Directory/File Location>

<(Optional) Fulfilled Requirements>

<(optional) Open Issues/Problems/Risks>

5.1.2 <Name black box 2

<black box template>

5.1.3 <Name black box n

<black box template>

5.1.4 <Name interface 1

...

5.1.5 <Name interface m

5.2 Level 2

Here you can specify the inner structure of (some) building blocks from level 1 as white boxes.

You have to decide which building blocks of your system are important enough to justify such a detailed description. Please prefer relevance over completeness. Specify important, surprising, risky, complex or volatile building blocks. Leave out normal, simple, boring or standardized parts of your system

5.2.1 White Box *<building block 1

... describes the internal structure of *building block 1*.

<white box template>

5.2.2 White Box *<building block 2

<white box template>

...

5.2.3 White Box *<building block m

<white box template>

5.3 Ebene 3

Here you can specify the inner structure of (some) building blocks from level 2 as white boxes.

When you need more detailed levels of your architecture please copy this part of arc42 for additional levels.

5.3.1 White Box <_building block x.1_

Specifies the internal structure of *building block x.1*.

<white box template>

5.3.2 White Box <_building block x.2_

<white box template>

5.3.3 White Box <_building block y.1_

<white box template>

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 Cross-cutting Concepts

Content

This section describes overall, principal regulations and solution ideas that are relevant in multiple parts (= cross-cutting) of your system. Such concepts are often related to multiple building blocks. They can include many different topics, such as

- domain models
- architecture patterns or design patterns
- rules for using specific technology
- principal, often technical decisions of overall decisions
- implementation rules

Motivation

Concepts form the basis for *conceptual integrity* (consistency, homogeneity) of the architecture. Thus, they are an important contribution to achieve inner qualities of your system.

Some of these concepts cannot be assigned to individual building blocks (e.g. security or safety). This is the place in the template that we provided for a cohesive specification of such concepts.

Form

The form can be varied:

- concept papers with any kind of structure
- cross-cutting model excerpts or scenarios using notations of the architecture views
- sample implementations, especially for technical concepts
- reference to typical usage of standard frameworks (e.g. using Hibernate for object/relational mapping)

Structure

A potential (but not mandatory) structure for this section could be:

- Domain concepts
- User Experience concepts (UX)
- Safety and security concepts
- Architecture and design patterns
- "Under-the-hood"
- development concepts
- operational concepts

Note: it might be difficult to assign individual concepts to one specific topic on this list.

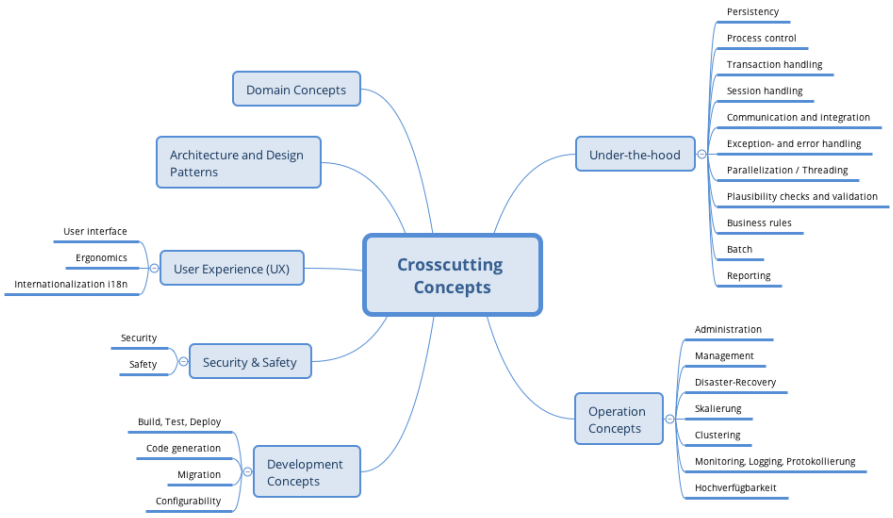

Figure 3: Possible topics for crosscutting concepts

8.1 *<Concept 1

<explanation>

8.2 *<Concept 2

<explanation>

...

8.3 *<Concept n

<explanation>

9 Design Decisions

Contents

Important, expensive, large scale or risky architecture decisions including rationals. With "decisions" we mean selecting one alternative based on given criteria.

Please use your judgement to decide whether an architectural decision should be documented here in this central section or whether you better document it locally (e.g. within the white box template of one building block).

Avoid redundancy. Refer to section 4, where you already captured the most important decisions of your architecture.

Motivation

Stakeholders of your system should be able to comprehend and retrace your decisions.

Form

Various options:

- List or table, ordered by importance and consequences or:
- more detailed in form of separate sections per decision
- ADR (architecture decision record) for every important decision

10 Quality Requirements

Content

This section contains all quality requirements as quality tree with scenarios. The most important ones have already been described in section 1.2. (quality goals)

Here you can also capture quality requirements with lesser priority, which will not create high risks when they are not fully achieved.

Motivation

Since quality requirements will have a lot of influence on architectural decisions you should know for every stakeholder what is really important to them, concrete and measurable.

10.1 Quality Tree

Content

The quality tree (as defined in ATAM – Architecture Tradeoff Analysis Method) with quality/evaluation scenarios as leafs.

Motivation

The tree structure with priorities provides an overview for a sometimes large number of quality requirements.

Form

The quality tree is a high-level overview of the quality goals and requirements:

- tree-like refinement of the term "quality". Use "quality" or "usefulness" as a root
- a mind map with quality categories as main branches

In any case the tree should include links to the scenarios of the following section.

10.2 Quality Scenarios

Contents

Concretization of (sometimes vague or implicit) quality requirements using (quality) scenarios.

These scenarios describe what should happen when a stimulus arrives at the system.

For architects, two kinds of scenarios are important:

- Usage scenarios (also called application scenarios or use case scenarios) describe the system's runtime reaction to a certain stimulus. This also

includes scenarios that describe the system's efficiency or performance. Example: The system reacts to a user's request within one second.
- Change scenarios describe a modification of the system or of its immediate environment. Example: Additional functionality is implemented or requirements for a quality attribute change.

Motivation

Scenarios make quality requirements concrete and allow to more easily measure or decide whether they are fulfilled.

Especially when you want to assess your architecture using methods like ATAM you need to describe your quality goals (from section 1.2) more precisely down to a level of scenarios that can be discussed and evaluated.

Form

Tabular or free form text.

11 Risks and Technical Debts

Contents

A list of identified technical risks or technical debts, ordered by priority

Motivation

"Risk management is project management for grown-ups" (Tim Lister, Atlantic Systems Guild.)

This should be your motto for systematic detection and evaluation of risks and technical debts in the architecture, which will be needed by management stakeholders (e.g. project managers, product owners) as part of the overall risk analysis and measurement planning.

Form

List of risks and/or technical debts, probably including suggested measures to minimize, mitigate or avoid risks or reduce technical debts.

12 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>*