Architecture development for a war game

Group 3:
Mark de la Court, Tycho Braams, Marko Groffen, Jan Boerman

11 November 2018

Abstract

Our goal for this project is to embrace an architectural mindset and gain an understanding of how an architecture is developed. To this end, an architecture for a war game is proposed. This is done in a structural manner, by identifying stakeholders, requirements, concerns and solutions in an iterative way. In this process, domain-specific techniques such as commonality-variability analysis and a SAAM analysis were applied. The result is a resilient and traceable architecture, as well as a valuable set of lessons learned about architecture development.

Index

Introduction

In this report, an architecture for a war game will be discussed to obtain an understanding of how architectures are designed. The architecture requirements are derived from the assignment, a market analysis and from the needs of the stakeholders involved. To ensure that the architecture is reliable and properly structured, it was carefully developed, mapping the solutions domains from concern domains, requirements and stakeholders. This was done in an iterative manner, to guarantee that the concerns, requirements, and solution domains also properly trace back to each other. The architecture was developed in such a way, that it is manageable and adaptable, keeping the tangling and scattering of concerns to a minimum.

Structure of the report.
First, the assignment for the game is discussed. Then, in order to obtain more requirements for the project and to put this assignment into context, a market analysis is conducted. This was to ensure that proper business requirements were in place and that the game has a unique value proposition to the players, on basis of which the rest of the requirements could be developed.

Next, the information obtained from the assignment and the market analysis was used to identify the stakeholders. This is done to ensure that the values of each of the stakeholders involved in the project are considered and can consequently be included in the requirements. Thereafter, requirements are engineered, deriving them from the perspective of the stakeholders and the value proposition of the product.

Then, the concerns are inferred from the requirements and they show problems that need to be overcome. How the concerns can be addressed is shown in the solution domains, to which the concerns can be mapped. First, a list of solution domains is discussed, which is then prioritised. The most important solution domains are worked out in detail, proposing a concrete architecture. Consequently, all solution domains are synthesized into one complete architecture which is then discussed.

Finally, the architecture is evaluated using the SAAM method, testing the resilience of the architecture to various scenarios that change the requirements of the game. Lastly, we discuss the project itself, what we learned and we present a conclusion.

The assignment

Computer Game
It is required to develop a computer game. The game should be a war game with the following elements: Vehicles and/or weapons; Natural terrain objects; Buildings and Roads/Paths.

Players should go through several stages to win the game. The ways to go through various stages is called games strategies.

Evolution
In the second version of the game, it has been realized that the game players would prefer to have a game with graceful degradation: During game playing, some elements of the game may get damaged. In this case, they may continue operating with reduced capacity until completely destroyed.

To this aim, some game elements had to be extended with historical information that influences their behaviour. This information can be various such as wear/tear of systems, damage of weapon systems, reduced fuel and battery capacity, etc.

It has been also realized that the system can still evolve in the future in similar ways. Therefore, the architecture must be made resilient to such evolutions as much as possible.

Exploration of the domain of and assumptions

Market analysis

The gaming industry is one that seems to experience an unstoppable growth. This year (2018) the industry grossed over $135 billion dollars in revenue, compared to only 70$ billion in 2012, and this growth is expected to continue, with an expected revenue of 180$ billion in 2021. This immense growth shows that investments in the gaming industry can be very profitable, not only immediately but also over the long term. In this assignment, the authors of this report have been hired by a game publisher to advise about the development of a war game and to develop the architecture for this game. The requirements from the game publisher have been given in the previous chapter. Since these requirements are very broad and abstract, a further refinement will be made based on the market characteristics. This is done because more concrete requirements are needed in order to make grounded architecture design decisions.

It is already given that a war game is to be developed. There are various types of war games, including co-op, team deathmatch, capture the flag and battle royale. Battle Royale is currently an upcoming genre, however, this market may be saturated because two leaders (Fortnite, PUBG) are already dominating the industry. Co-Op games are the second most popular game genre at this time, after Battle Royale ***[1]***. Co-Op games rely heavily on game strategies, which is a requirement for the game to be developed, and they have proven to be very profitable, for example, popular Co-Op games are Call of Duty, Borderlands and Far Cry. Therefore, it has been decided that the game to be developed is of the type Co-Op. In a co-op kind of game, several players will work together usually using online multiplayer, like for the examples mentioned earlier, to defeat AI players in various stages or ‘levels’.

Online multiplayer games have proven to be very popular and successful. The use of online multiplayer promotes engagement, as players will convince each other to play the game together. Online multiplayer games can also be considered to be very profitable, a study in Latin America in 2016 showed that over two-thirds of all profits from the gaming industry came from games that include some form of online multiplayer ***[2]***. Therefore, developing a game that supports online multiplayer, especially considering the kind of war game that is being developed, would be a reasonable choice.

Additionally, a choice of platform has to be made. Virtually all game publishers publish their games for multiple platforms. the choice is not so much whether the game will be developed for multiple platforms, but for which set of platforms the game will be developed. Since mobile gaming has recently outgrown PC and Console gaming, adding support for mobile platforms would be a logical choice also considering future growth of this market. The desktop/console gaming market is expected to grow 2-3% from 2012 to 2021, while the gaming market is expected to see a 21% growth ***[3]***. This shift is logical, since mobile devices are becoming increasingly powerful, and since game development tools increasingly

support more platforms. This makes releasing a game to more platforms, including mobile platforms, easier and more common.

Finally, the monetization of games is to be considered. Traditionally, games made money using a pay to play business model, however, increasingly a free to play model is adopted in the industry, and it has proven to be very popular. Over 60% of all revenue in the gaming industry is currently generated by games adopting the Free to Play monetization model ***[4]***. A free to play business model knows many advantages, including a lower threshold for users to use the game and a continuing revenue after the user has decided to use the game.

Positioning

Cross-Platform Multiplayer
Unlike existing co-op games, the game to be developed will feature cross-platform multiplayer. This allows the player to play the game with their friends, regardless of which platform they are using. This will also allow for faster adoption and popularity of the game as friends may convince you to play it, and there is no threshold of switching platforms to join. The use of cross platforms has a major impact on the architecture, as multiple platforms and multiple multiplayer protocols have to be supported.

Co-Op Play Mode
As argued, the game will feature a co-op playing mode, as it is the best fit for the given requirements while still being a proven way to generate revenue. While other co-op games have various stages that have to be completed, this game will also offer multiple strategies for progressing through these stages. For example, other than shooting, which is usually the only game strategy available in co-op games, users may also apply other game strategies like driving or tactical defence. This impacts the game architecture, as not only weapons but also other means of progressing through stages have to be supported.

Free to Play
The business model used impacts the architecture significantly. In a free-to-play game, additional content, like weapons and game stages but also cosmetic items like clothing and armour may be obtained by means of purchase. Hence, it should be possible to dynamically add these items to the game on runtime after a purchase has been made.

Stakeholders

S1 Players: The players are the final end users of the software. As such, they could indicate features they would like to see. They'll also have certain expectations about the type of system the game can run on, the performance of the game and the way controls are implemented.
Players are often grouped using player profiles. These describe the way players play a game. These should be used to ensure the game is viable for all the intended players.

S2 Latency stakeholder: Since the game will be played across a network, latency is an important stakeholder. Latency influences the response time of the game to input from the player, hence latency is important as it significantly impacts the end user experience. It also influences the information that can or should be sent across the system and how much should be known by the client side. As a high latency can cause an unintended increase in difficulty, latency is an important stakeholder throughout the design process.

S3 Graphics Expert: Games heavily rely on the graphics that they contain. These graphics should be consistent no matter the state of the game. Graphics will also be used to control the game with menus, provide information to the player about the state of the game, for example, a graphic showing the equipped weapon and it's ammunition.
The graphics will require information from several parts of the architecture and will be the link between the player and the game. As such, it is important to take this into account when designing the architecture.

S4 Cross-platform expert: Based on the market analysis, it was decided that the game should be cross-platform. This will require that the implementation can run on multiple hardware setups. Furthermore, the players should be able to play together, no matter the hardware they are using. The cross-platform expert should ensure that the chosen architecture can be deployed to allow these features.

S5 Developer: The developers will have to implement a program that conforms to the created Software Architecture. It is therefore important to take into account what technologies the developers are familiar with when deciding on architectural constructs.

S6 Gameplay Expert: The game will be players against artificial enemies. It is important to include the requirements for artificial intelligence early in the architecture. A Gameplay expert should also monitor if the different choices that are possible for the player are balanced. If one type of choice leads to a much better or easier result, the other choices will become irrelevant.

S7 Consistency stakeholder: An important aspect of the game is the cooperation with other players. For this cooperation to be effective, it is important that all players have the same view of the game state. If for example, an enemy is present at different locations for different players, it becomes impossible to have consistent consequences for the actions of these players. The architecture should make sure that the server and players all use the same game state.

S8 DLC Expert: Since the game will be available as a free product, the way to make the game profitable will be through extra payments in the game. These can be custom skins for players or weapons etc. Another way is to have payments for extra content. This often requires external payments and downloading extra software modules representing in-game items or stages. The architecture needs to accommodate ways to include this.

S9 Scalability stakeholder: Players want to be able to play the game, no matter how many other players are currently playing the game. While the architecture is designed, the capacity stakeholder should ensure that bottlenecks are avoided whenever possible. He should also ensure that the architecture allows for additions of extra capacity when the popularity of the game increases.

S10 Adaptability stakeholder: As mentioned in the assignment description, a second version of the game will include changes in the degradation of objects. Further changes are also expected. The design of the architecture should ensure that such extensions can be made to the game without requiring a redesign of the architecture.

S11 Security Expert: Since players will be making payments for extra features in their game, these payments should be secure. Furthermore, the architecture should make sure that the features obtained are safely managed. The architecture should make it hard for third parties to steal from or hijack an account of a player.

S12 Distributed Computing Expert: To reduce the load on the network, at least part of the computing for the game will take part in the client on the player's hardware. Furthermore, to accommodate players from all over the world, it will be helpful to distribute the server-side computing. The expert can help indicate where the challenges will be and how this should be incorporated in the architecture.

S13 Scenario Designer: The scenario designer will create the levels that the players will be challenging. It is likely that there will be a lot of reuse of objects within different levels. As such it would be more efficient to incorporate the creation of these levels within the architecture. This requires a collection of elements, and an editor to combine these elements. It should also be possible to define how the scenario changes based on the number of players that are cooperating.

The requirements

In the section below, the various requirements of the system are discussed. They are subdivided into quality, functional and system requirements. First, we will discuss the business requirements, that focus on the monetary side of the system. These are not direct architectural requirements, but they are nonetheless important for the overall system and they provide context to the other requirements.

Code	Requirement	Stakeholders
Business Requirements		
R1	The game should be monetizable	S8
R2	The implementation should not exceed the budget	S2, S3, S4, S7, S8, S9, S10, S11

The quality requirements describe the performance requirements of the game, mainly focussing on playability. The functional requirements describe the functional behaviour to be included. And finally, the system requirements apply to the overall functioning of the system. A brief motivation for each requirement is included in a separate list.

Code	Requirement	Stakeholders
Quality requirements		
R3	The game should be playable with a consistent framerate	S2, S12
R4	The graphics should offer a consistent and realistic view of the game world	S3
R5	The game should accommodate smooth movements after controller input	S2, S3
R6	The connection between players should be secured	S11
R7	Players should not be able to cheat to keep games fair	S6, S11
Functional requirements		
R8	The game should incorporate weapons, vehicles, natural objects and roads	S13
R9	The game should include spawning AI enemies to fight against	S6
R10	The game is in 3D	S3

R11	The game has to include wear and tear of game objects	S13
R12	The game should have multiple levels of varying difficulties that can be won	S1, S6, S13
R13	The game should offer choices to a player, so they can use different strategies to progress through stages	S1, S6, S13
R14	The server should remember the progress/game state of players. Overall progress should be stored and progress in a stage can be saved at specific moments	S7
R15	The player should be able to choose graphical settings so that the game can run on the available hardware	S1, S3, S4
R16	The game should have a graphical interface to start new games	S3
R17	The game should be controllable using the preferred input of the platform	S1, S6
R18	The game should support downloadable extensions in the form of new stages, new weapons/vehicles and cosmetic items	S6, S8, S13
System requirements		
R19	The game backend should be scalable	S9, S10, S12
R20	The game should run on multiple platforms: PS4, Xbox One, PC, Android, Nintendo Switch	S4, S5
R21	The game client should be playable cross-platform	S4, S5
R22	The game should include a secure connection whenever the user wants to buy expansions	S8, S11
R23	The architecture should be resilient to evolutions and further extensions	S10

Motivations:

Code	Motivation
R1	In order to at least be repaid for the development costs, the game should contain an in-game mechanism to make money. It has to be in-game since the basic game is provided as a free-to-play product to users.
R2	In order to produce the game, a budget limit will be set. This limit should not be exceeded.

R3	Changing frame-rates are often considered to be annoying by users, so the game should provide a steady frame-rate.
R4	Glitching graphics and objects or otherwise unrealistic graphics or behaviour of objects heavily decrease the engagement and playability of the game and should, therefore, be absent.
R5	Lag between the input of a player and the execution in-game can make the game unintentionally very difficult, so this lag should be kept to a minimum.
R6	It should not be possible to breach the computer of a user or disrupt the connection between two players due to an insecure connection.
R7	In order to keep the gameplay fair, cheating should not be possible. Additionally, if players can cheat to obtain certain items or goals, they may be less inclined to make in-game purchases to reach the same objectives.
R8	In order to allow interaction with the in-game environment, the game should contain at least these objects.
R9	Since the game is a co-op shooter, the players need enemies to fight against. These will be provided in the form of AI controlled enemies.
R10	To give a realistic battle experience, the game will be made with 3D graphics.
R11	To improve interaction with the in-game environment, map objects will be sensitive to damage dealt by either the player or AI enemies. Also, the weapons used by players will have limited ammunition. Furthermore, the players and AI are susceptible to damage.
R12	The game will be divided into separate levels/stages. These levels have a simple win or lose objective: the players either succeed or have to retry. Each level will have a different game environment and a different set of enemies.
R13	To enforce the strategic challenge within the game, the in-game environment should allow for different paths that can be taken. Each path will have its own pros and cons when it comes to beating the game. Some of these paths will be more strategic (which route to take for example), where others are maybe more physical (reaction speed for example)
R14	Certain checkpoints will be built in the game to save the player's progress up to that point within a level. The overall progress through the stages should also be saved for players separately. The progress information should be accessible online since it is needed for matchmaking between players.
R15	To make the game compatible with various hardware setups, the players should have the ability to change the level of in-game graphics like the resolution of textures or screen resolution.

R16	In order to start a new game or resume from previous progress, the player should be presented with a user interface. This user interface can also implement a settings components, like the graphical setting required in R15.
R17	It should be possible for the user to control the game. The input method depends on device-specific hardware, such as a touchscreen, mouse or gamepad.
R18	The game should allow for extra content to be added to the game after installation. These can allow the player to customise their character, or to unlock and download extra content such as levels or weapons.
R19	If the popularity of the game increases, there will be a larger demand on server connections. To manage this, both the servers and game code should allow for expansion to satisfy such demands.
R20	In order to reach a large audience, the game should be published on the most popular game platforms.
R21	In order to reduce the development time and costs, the code for the game should be easily adaptable for the various platforms. This makes sure it is not necessary to write code for 5 different games. It should also make cross-platform connecting easier.
R22	Since users will need to be able to make a monetary transaction, personal and sensitive data is involved. Therefore, a secure connection is required to make such a transaction safe.
R23	The architecture should be resilient to evolutions in the architecture due to e.g. feedback from one of the stakeholders.

The concerns

We group concerns into multiple categories, namely Player concerns, Computer science concerns, Quality concerns, Math concerns and Separation of concerns.

Player concerns

Player concerns are problems that directly affect the end users of the game.

Code	Concern	Req
C01	**How can we model the in-game user interface?** There must be an intuitive interface. For each platform that the game is released on, indicators such as those for health bars, minimap and ammunition should render in a convenient place.	R10, R16
C02	**How can we model user input in our architecture?** There should be a way for users to control the game's interface as well as their in-game character. The game should consequently respond to the user's input.	R17
C03	**How can we model cross-platform gameplay in our architecture?** There must be cross-play between platforms so players can play with friends that use different platforms.	R20, R21
C04	**How can we model game saving and loading for online and offline sessions?** Games can be saved and loaded. This concern is a quality problem too because loading and saving should be fast and seamlessly work with online as well as offline sessions.	R12, R14

Computer science concerns

Computer science concerns are algorithmic problems.

Code	Concern	Req
C05	**How can we model the game components?** Game components, such as game stages ('levels'), strategies, worlds, and use items (weapons, skins, vehicles), should not be fixed and retrievable at runtime.	R8 R12, R13, R18
C06	**How can we model network latency in our architecture?** Players should experience little latency in multiplayer mode. This is a networking issue that requires a well-designed protocol that uses as little bandwidth as possible, but it's also a problem of finding players that are close to each other geographically.	R5, R14
C07	**How can we model the scaling of resources to fit the number of players, in our architecture?**	R9, R19

	The game should scale to large numbers of players and bots. The game should use algorithms that are efficient in terms of time and space complexity.	
C08	**How should we model our architecture to allow for secure communications?** The protocol must be secure. Player's connections cannot be hijacked and players' identities cannot be spoofed. Furthermore, any personal data of player's accounts should not be sent to clients	R6, R22
C09	**How can we include cheating mitigations in our architecture?** Cheating should be mitigated. Cheating is a problem that occurs in many online games and should be mitigated to ensure a good player experience. Common cheats are aim-assistance and faster movement.	R7

Quality concerns

Problems considering performance or otherwise non-functional.

Code	Concern	Req
C10	**How can we model extendability of the architecture?** The architecture should be extendable in the future.	R23
C11	**How can we model the consistency of the game state across all servers and clients in our architecture** The game must run smooth and consistently on all platforms. Apart from networking latency, the game's internal memory layout and visuals should be optimized per platform.	R3, R4, R10, R15

Math concerns

Problems of a mathematical nature.

Code	Concern	Req
C12	**How can physics be modelled in our architecture?** Physics in the game should be accurate, similar to the real world. Calculating where game objects should be rendered on the screen is also a part of this. Some of these calculations are computationally expensive, so this is also a quality concern, we must look for techniques to minimize the number of CPU operations.	R4, R10
C13	**How can we model Artificially Intelligent computer players in our architecture?** Computer players in the games should have some form of AI, making them a challenge to beat, but not impossible. Computer players should be able to make decisions and have strategies.	R9

Separation of concerns

Problems regarding inevitable scattering or tangling of concerns.

Code	Concern	Req
C14	**How can we model graceful degradation in our architecture?** Actions in the game world, such as a vehicle hitting a tree, should influence the state of in-game objects.	R11
C15	**How can we model in-game purchases in our architecture?** In game units, such as certain stages, worlds and user items (i.e. a vehicle or skin) should only be available for download and use after a transaction.	R1, R18

Identification and exploration of the solution domains

Game Engine

A game engine is an environment that can be used to develop computer games. They can be responsible for rendering 2D/3D graphics, the physics of a game, the sound, scripting and animation. Existing game engine configurations are sometimes reused or adapted to create new games. Game engines often also help the developer in making it easy to deploy to multiple platforms. With the popularity of First-person shooter games, engines for these games are often seen as a separate category, which is at the forefront of developing technologies.

References:

- Gregory, J. (2009) Game Engine Architecture. https://www.gameenginebook.com/
- Nilson, B. and Söderberg, M. (2007) Game Engine Architecture. Mälardalen University. http://citeseerx.ist.psu.edu/viewdoc/versions?doi=10.1.1.459.9537
- Caltagirone, S. Keys, M. Schlief, B. Willshire, M.J. (2002) Architecture for a massively multiplayer online role playing game engine. Journal of Computing Sciences in Colleges, Volume 18 Issue 2, December 2002, 105-116 https://dl.acm.org/citation.cfm?id=771339

Physics

Most 3D games, especially our game, make use of a physics engine. A physics engine is a piece of the game engine that calculates locations and velocities of game objects in 3D space. The purpose is to make the game look realistic; usually by making objects behave as they do in the real world, according to the classical Newtonian laws of physics. The most interesting part of physics is what will happen when objects in the game collide. Off-the-shelf game engines often include a physics engine. Existing engines solve this problem using built-in ‘materials which can be set for objects. The materials define parameters for the collision behaviour of the object. Those parameters can be modelled using object-oriented design patterns.

Physics references

- Choi, J., Shin, D., & Shin, D. (2006). Research on Design and Implementation of Adaptive Physics Game Agent for 3D Physics Game. In Agent Computing and Multi-Agent Systems (pp. 614–619). Springer Berlin Heidelberg. https://doi.org/10.1007/11802372_68
- Hummel, J., Wolff, R., Stein, T., Gerndt, A., & Kuhlen, T. (2012). An Evaluation of Open Source Physics Engines for Use in Virtual Reality Assembly Simulations. In Advances in Visual Computing (pp. 346–357). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-33191-6_34
- Unity Technologies. (2018, October 10). Physic Material. Retrieved November 8, 2018, from https://docs.unity3d.com/Manual/class-PhysicMaterial.html

- Unreal Engine. (n.d.). Retrieved November 8, 2018, from https://docs.unrealengine.com/en-us/Engine/Physics

Artificial Intelligence

Artificial intelligence is responsible for the actions performed by all the characters in the game that are not players. Artificial intelligence is sometimes seen as part of the game engine, but since it is almost always unique for a game, it can be considered as a separate component.
Game AI is usually used to refer to a broad set of algorithms that are used to compute the actions of Non-player characters (NPC). This could also be described as automated computation, as things like computer learning are usually not included in game AI. Recent efforts are starting to bring Game AI closer to actual AI.

References:

- Millington, I. John Funge. (2009) Artificial Intelligence for Games. https://www.taylorfrancis.com/books/9780080885032
- Lu, F. Yamamoto, K. Nomura, L.H. Mizuno, S. Lee, Y. Thawonmas, R. (2013) Fighting game artificial intelligence competition platform.2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE) DOI: https://doi.org/10.1109/GCCE.2013.6664844
- Yannakakis, G.N. (2012) Game AI revisited. Proceedings of the 9th conference on Computing Frontiers, 285-292. DOI: https://doi.org/10.1145/2212908.2212954

Platforms

‘Platforms’ are the solution to our cross-platform concern. To support cross-play, all platforms use the same protocol for communication. The hardware and operating systems of platforms are actually unimportant as long as they support saving and loading bytes to disk and sending bytes over the internet - which all modern platforms do. Some off-the-shelf game engines already support multiple platforms out of the box. If you don’t opt for such an off-the-shelf solution some platform-specific code is needed for those platforms not supported out of the box.

To make the server software scale, we need a kind of orchestration solution. An example of an architecture could be one where game clients contact the orchestrator and are forwarded to actual game servers. The orchestrator can then spin up new game servers if necessary.

References

- Gambetta, G. (n.d.). Fast-Paced Multiplayer (Part II): Client-Side Prediction and Server Reconciliation. Retrieved November 8, 2018, from http://www.gabrielgambetta.com/client-side-prediction-server-reconciliation.html
- Xu, Y., Nawaz, S., & Mak, R. H. (2014). A Comparison of Architectures in Massive Multiplayer Online Games.

- Hsu, C. A., Ling, J., Li, Q., & Kuo, C.-C. J. (2003). The design of multiplayer online video game systems. In A. G. Tescher, B. Vasudev, V. M. Bove, Jr., & A. Divakaran (Eds.), Multimedia Systems and Applications VI. SPIE. https://doi.org/10.1117/12.512201

Protocols

The client and the server will communicate with each other using certain protocols. Since this is a live multiplayer game, the protocols need to be not only reliable but also allow for high data loads with low latency. The number of protocols that can be used may be restricted, since consoles such as Xbox and PlayStation may force developers to use dedicated platform-specific protocols. It should, therefore, be ensured that protocols are selected that support a broad range of hardware.

Protocols References:

- Fiedler, S., Wallner, M., & Weber, M. (2002). A communication architecture for massive multiplayer games. In Proceedings of the 1st workshop on Network and system support for games - NETGAMES ’02. ACM Press. https://doi.org/10.1145/566500.566503
- Lopes, C. (2011). Hypergrid: Architecture and Protocol for Virtual World Interoperability. IEEE Internet Computing, 15(5), 22–29. https://doi.org/10.1109/mic.2011.77
- Inoue, K., Pasetto, D., Lynch, K., Meneghin, M., Muller, K., & Sheehan, J. (2013). Low-latency and high bandwidth TCP/IP protocol processing through an integrated HW/SW approach. In 2013 Proceedings IEEE INFOCOM. IEEE. https://doi.org/10.1109/infcom.2013.6567108

Database

The database is where gameplay and progress are saved. However, also other persistent information like user details, rankings, and game units will be stored in the database. A database will be integrated into the architecture. While the use of databases is common across many types of applications, the fact that a database is needed on both the client side and the server side, and the fact that the server side database should be very scalable may bring challenges. Therefore, distributed and scalable databases should be considered.

References database:

- Özsu, M. T., & Valduriez, P. (2011). Principles of Distributed Database Systems, Third Edition. Springer New York. https://doi.org/10.1007/978-1-4419-8834-8
- Bowen, T., Gopal, G., Herman, G., & Mansfield, W. (n.d.). A scalable database architecture for network services. In International Symposium on Switching. IEEE. https://doi.org/10.1109/iss.1990.765806
- Delis, A., & Roussopoulos, N. (1992, August). Performance and scalability of client-server database architectures. In VLDB (Vol. 92, pp. 610-623).

Game Units

Units in our game should be extendable. The number of types of units in our game is undefined so that it is possible to add more in the future. To support this, we can make use of features that are commonly used in games that support 'mods'. Custom units can be implemented easily if they are 'data-driven'. This data consists of the textures of the units, but also their physics behaviours and interactions with other objects and players. Such game unit data can easily be retrieved from our game servers, and the data can be interpreted using the interpreter pattern. Other techniques to support custom units are scriptability, where the data of the unit includes a script that uses a fixed API and describes the behaviour of the unit. The engine then interprets the script making its behaviour come to life. The last technique listed here is, if our game will be implemented in an object-oriented language, it is possible to load classes dynamically from over the network and create instances dynamically at runtime.

References:

- Passos, E. B., Sousa, J. W. S., Clua, E. W. G., Montenegro, A., & Murta, L. (2009). Smart composition of game objects using dependency injection. Computers in Entertainment, 7(4), 1. https://doi.org/10.1145/1658866.1658872
- I want to make a moddable game. How does this affect my programming language choice? (n.d.). Retrieved November 8, 2018, from https://gamedev.stackexchange.com/questions/21819/i-want-to-make-a-moddable-game-how-does-this-affect-my-programming-language-cho
- Nystrom, R. (n.d.). Component. Retrieved November 8, 2018, from http://gameprogrammingpatterns.com/component.html

Transactions

In order to make the game monetizable, the game includes buyable expansions to the game. These are in the form of extra game levels, weapons and skins (appearances of either weapons, characters or vehicles). This requires the possibility of transactions within the game. This can be organized by introducing an in-game currency, which can, in turn, be spent on the extensions. Another option is the direct purchase of game elements in the form of downloadable content (DLC). In the latter case, the content is downloaded and immediately playable within the game. The purchased extensions can be either platform specific or synchronized between platforms and based on the player's account. In order to purchase the items, either the transaction mechanisms of the various platforms can be used (e.g. PlayStation store, android market) or a separate transaction mechanism is created within the game. In the second case, security is the main focus.

References:

- Tomic, N. (2017). Effects of micro transactions on video games industry. Megatrend Revija, 14(3), 239–257. https://doi.org/10.5937/megrev1703239t
- Oh, G., & Ryu, T. (2007, September). Game Design on Item-selling Based Payment Model in Korean Online Games. In *DiGRA Conference* (pp. 650-657).

- Holin Lin, & Sun, C.-T. (2011). Cash Trade in Free-to-Play Online Games. Games and Culture, 6(3), 270–287. https://doi.org/10.1177/1555412010364981

Input

To control the game characters and environment, the player gives input in the form of a control mechanism, which is translated to the in-game world. Since the game is distributed over existing consoles and gaming devices, often a standard input is already delivered by the platform creator. This is a keyboard for PC or a controller for the console platforms. Key mapping is here variable, but to provide a short learning curve, the generally used key mapping for first-person shooters should be present. Making the key mapping customizable to the player, however, is a good option. The mobile phone is an exception since the touchscreen does not have a predefined control interface. Most competitors that created first-person mobile games map a controller interface onto the touchscreen, mainly to stimulate recognition with the player. Research shows that a natural mapping of controls does increase the engagement in the game. So the possibility to introduce a game-specific controller is there, to provide a more realistic alternative to the other game controllers. It is also important that the reaction time between the input and the in-game reaction is short since a instant feedback to the player increases the enjoyability of the game. This can be solved by the game code handling the input signal.

References

- Heatherly, B. & Howard, L. (n.d.) “Video Game Controllers.” Retrieved November 8, 2018, from http://andrewd.ces.clemson.edu/courses/cpsc414/spring14/papers/group2.pdf
- Skalski, P., Tamborini, R., Shelton, A., Buncher, M., & Lindmark, P. (2010). Mapping the road to fun: Natural video game controllers, presence, and game enjoyment. New Media & Society, 13(2), 224–242. https://doi.org/10.1177/1461444810370949
- Klimmt, C., Hartmann, T., & Frey, A. (2007). Effectance and Control as Determinants of Video Game Enjoyment. CyberPsychology & Behavior, 10(6), 845–848. https://doi.org/10.1089/cpb.2007.9942

The traceability matrix

Requirements and stakeholders

This matrix links the requirements to the stakeholders.

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13
R1								X					
R2		X	X	X			X	X	X	X	X		
R3		X										X	
R4			X										
R5		X	X										
R6											X		
R7						X					X		
R8													X
R9						X							
R10			X										
R11													X
R12	X					X							X
R13	X					X							X
R14							X						
R15	X		X	X									
R16			X										
R17	X					X							
R18						X		X					X
R19									X	X		X	
R20				X	X								
R21				X	X								
R22								X			X		
R23										X			

Concerns and requirements

This matrix links the concerns with the requirements.

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
R1*															X
R2*															
R3											X				
R4											X	X			
R5						X									
R6								X							
R7									X						
R8					X										
R9							X						X		
R10	X										X	X			
R11														X	
R12				X	X										
R13					X										
R14				X		X									
R15											X				
R16	X														
R17		X													
R18					X										X
R19							X								
R20			X												
R21			X												
R22								X							
R23										X					

** Note that R1 and R2 are business requirements, which may not be directly reflected by the concern domains.*

Solution domains and concerns

This matrix shows how the solution domains map onto the concerns.

	Game Engine	Game Units	Client-Server Platforms	Artificial Intelligence	Transactions	Input
C01	X					
C02						X
C03	X		X			
C04			X			
C05		X				
C06			X			
C07			X			
C08			X		X	
C09			X			
C10		X	X			
C11	X					
C12	X					
C13				X		
C14	X	X		X		
C15					X	

Prioritization of the solution domains

Prioritised domains

Solution domains are prioritised such that it can be estimated which parts require the most attention when developing the architecture. The most important solution domains identified here will be analysed further in the individual studies to identify a more concrete architecture for the most significant solution domains. The following solution domains were identified as most important: Game Engine, Platform, Game Units, AI.

Platform and Game Engine are important central architectural components of any game. How they are modelled significantly affects the performance, security and adaptability of the product. The platform design poses an extra challenge considering the cross-platform requirements imposed on this game.

Game Units are relevant for most complex open world games, and especially for this game. Complex and engaging environments require a well-designed model for game units. Since game units must be downloadable and instantiable on runtime, this imposes restrictions on the architecture that will have to be considered. A proper architecture for game units will allow for a more adaptable game world, where new elements can be integrated into the game with ease without making major changes to the architecture.

AI plays a larger and more central role in this game than is usual in most games, as the main goal of the game is to defeat AI opponents rather than other players. To create an engaging game experience, the AI should be able to demonstrate a wide variety of behaviours to be an interesting and complex opponent that can be defeated. To accomplish this, an AI architecture is proposed that allows for these complex and integrated behaviours, and that can be adapted to show different behaviours in future revisions of the game.

The importance of the Game Engine, Platform and Game Units solution domains also follows from the traceability matrix, which shows how they are related to the various concern domains.

Demoted solution domains

Unfortunately, not all solution domains could be analysed in detail. These solution domains still need a proper architecture for the game to succeed, but these domains are architecturally less challenging and do not contribute to the success of the product in the same way as the prioritised domains do. Input and transactions, but also physics, databases and protocols (as sub-architectures of other solution domains) will therefore not be discussed in further detail in this report. However, physics, databases and protocols will be briefly discussed in the context of their parent’s architecture.

Detailed analysis of solution domains

Solution Domain: Artificial Intelligence

Solution Domain description by Mark de la Court

Solution domain description

Artificial Intelligence plays a key role in co-op games since the game is centred around defeating computer-controlled opponents. These opponents are usually called non-player characters, or NPC's. The purpose of this domain is to introduce an architecture that allows for intelligent game units, with varying behaviours, that can interact with the player and the environment. A game AI is different from the 'Pure AI' that one would use to solve real-life problems. While with a Pure AI its purpose would, for example, be to create an undefeatable computer player, the challenge with game AI is to create a believable component that interesting and challenging to defeat.

Current developments and trends

There are two implementations for Artificial Intelligence in gaming, centralized and distributed intelligence. Traditionally, AI was implemented as a centralized architecture, with a single artificial intelligence component that controls all NPC's. However, with the increase in complexity in games and NPC behaviours, such centralized architectures are barely used anymore. The current state of the art is to use *decentralised intelligence*, where each NPC or agent has its own intelligence and is able to make its own decisions somewhat independently of all other agents. Such a decentralised architecture with individual agents that individually observe and act on the environment is referred to as an *Agent Architecture*.

Agent architectures have a few things in common. They all have some mechanism that *perceives the world*. They then *process their perception* of the world, and finally, they *act on the world*, just like a regular playable character would. Agent architectures most fundamentally differentiate from each other in the way that they process their perception and decide on actions. Russell & Norvig (2003) categorise agents into five classes.

Simple reflex agents - These agents act on the environment they currently perceive based on a set of simple rules that output actions when certain conditions are met. These agents can be considered unsuitable for open world games, as they require a complete perception of the world.

Model-based reflex agents - These agents act on an internal model of the world, which is continuously updated by their perceptions. These agents may also be aware of the impact of their actions on the world, as the model based reflex agents take into account the actions that happened in the world.

Reflex agents, both simple and model-based, act often work on a complex rule database, for example, they could use decision diagrams, behavioural trees or (hierarchical) state machines to determine the next action.

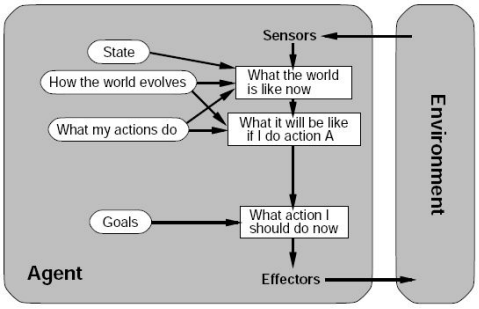
Goal-based agents - These agents work similar as model-based reflex agents, however, they do not select their action based on rules, but instead based on internal goals they may have, such as reaching a certain location. They will select those actions that best fit their goals. Goal-based agents are seen as more flexible and easier to maintain than reflex agents, especially when there are many different goals and actions. Goal-based agents are also called planners.

Utility-based agents - Utility-based agents refine upon goal based agents, by not only distinguishing between goal states and non-goal states but instead by assigning a certain utility to a goal state. The utility-based agent can, therefore, have multiple goals, and decide which goals are the most desirable to obtain.

Learning agents - Learning agents interpret the results of their previous actions and then learns how it could do better in the future. It has an element that generates new actions based on those experiences, and a performance element that selects the best action based on what was learned and based on the model of the world. This kind of agent is currently a popular research topic. Combined with techniques such as machine learning, these agents can be very powerful for a broad range of tasks. For gaming, however, they are seldom used. Agents may often react in unpredictable ways and they may not adhere to specific behaviours resulting in non-convincing NPC experience and low designer control.

Of these different agents, Goal-based agents are the most fitting for the game that is to be designed. They allow for a complex and a large set of behaviours that are still easy to manage, which is not possible with reflex based agents where every sequence of behaviours has to be hardcoded. Goal-based agents are very adaptable, as goals and actions can be added and removed easily, without having to adjust any sequences or states machines of behaviours. And they are easy to maintain, unlike Utility-based agents which are hard to design, edit and tune[1].

Commonality & variability analysis



While all goal-based agent architectures will in general terms adhere to the high-level description depicted in the image on the right, their specific implementations are different. The commonalities and differences between implementations are discussed in this section.

Sensors

As stated earlier, an agent has to perceive the world. However, if agents would need to process the entire world continuously this would be very inefficient. Agents can instead observe the world once to create an internal model and then

[1] http://intrinsicalgorithm.com/IAonAI/2012/11/ai-architectures-a-culinary-guide-gdmag-article/

update this model using some event-driven system (Cristea et al., 2011). The agent is notified about any changes in the world such as the actions of a player or NPC nearby, or changes in the state of another game unit. These event systems are usually integrated into the platform, and the agent will need to subscribe to receive any relevant events nearby.

Controller

The controller manages the AI player by means of actions and goals (Miyake, 2016). Each action usually has three components, the action itself, which is the effector on the world, the preconditions to start the action and the impact of the action, for example obtaining a certain goal. Additionally, a list of goals is defined for each NPC. These goals may have preconditions, for example, the goal to patrol a certain area may only be activated if no player is nearby.

Rather than using programming objects to represent actions and goals, it is more common to use a language to define these goals and actions. This way, actions and goals can be changed dynamically, without changing the architecture. There are two popular planners (Dicken, 2012):

Planner	Based on	Description	Example Games
Goal Oriented Action Planning (GOAP)	Standford Research Institute Problem Solver (STRIPS)	Searches through (possible) world states by applying actions. Typically this search is done backwards from the goal state. Works very well in open world games, but not very well in scripted environments were specific actions are desired.[2]	Tomb Raider (2013), Just Cause 2, F.E.A.R series
Simple Hierarchical Ordered Planner (SHOP)	Hierarchical Task Network (HTN)	HTN planners are hierarchies of tasks that can be broken down recursively. Goal tasks are broken down in compound tasks which are then broken down in primitive tasks. Provides more control, but requires more complex definitions.[3]	Killzone series, Transformers: War for Cybertron

Managers

The controller controls the high-level behaviour of the AI, for example, the GOAP controller determines the goals and actions, however, the AI also needs other components to manage more concrete behaviours such as finding the best path to take, reloading a weapon, aiming, and keeping track of a target. A list of such managers found in various systems (Long, 2007) is given below.

Manager	Description
Navigation	Each NPC that is able to move needs a pathfinding mechanism. Actions can include the movement to a certain location or following a certain path, and the pathfinding

[2] http://aigamedev.com/open/review/planning-in-games/
[3] https://www.researchgate.net/publication/261217494

	mechanism will provide the most efficient path to this location. Usually, A* pathfinding algorithm is used for this, as it able to quickly find the path with the lowest cost to a certain destination.
Weapons	Weapons need to be managed, for example, guns may need to be reloaded, and swapped, and the AI needs to aim at a certain target with a certain accuracy. The weapons manager takes care of this.
Team	The NPC may operate in a team, or otherwise together with others NPCs. The team manager tracks other team members and modifies the space of actions (such as paths taken or tactics used) accordingly.
Target	The target manager tracks the targets, and decide which target to focus on.
Degradation	The degradation manager tracks the health of the NPC and may adjust available actions and goals accordingly.

Blackboard
The various managers proposed above need to communicate with each other. For example, the target that the agent is tracking will influence how the weapon is aimed and what the cost is to navigate a certain path (an enemy can make a path more costly due to possible injury when nearby the enemy). To facilitate communication between the components of an AI, an internal blackboard can be introduced through which components can communicate (Corkill, 2003). In our example, this means that the target manager will post information to the blackboard, which the navigation and weapons manager can then obtain from the blackboard, such that they do not both have to communicate directly with the target manager. A blackboard may however also limit the exchange of specific information between to components, as the information communicated has to be adjusted to a specific format to be supported by the blackboard.

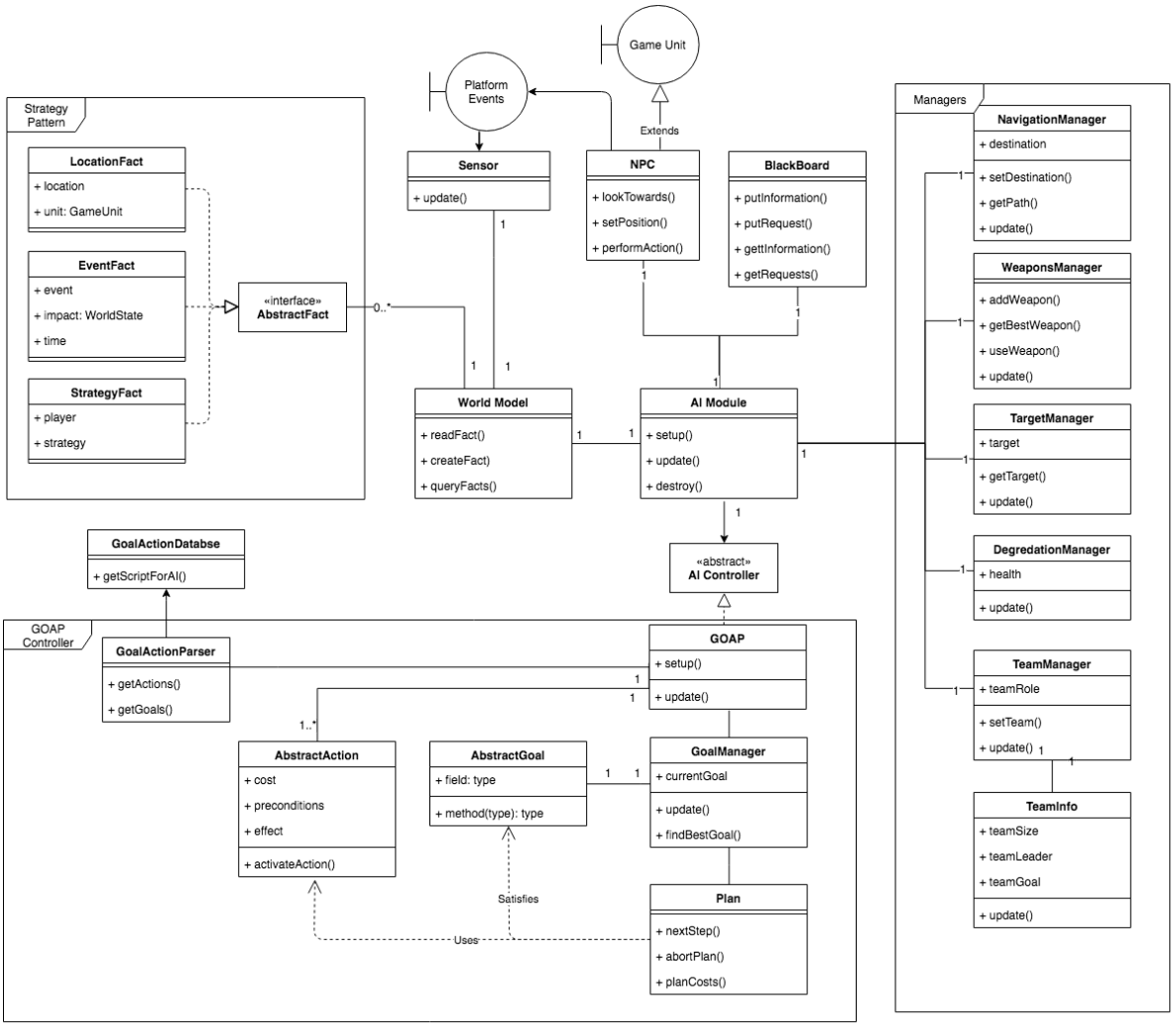
Constraints among alternatives

The alternatives and modules discussed in the commonality and variability analysis may be constrained by each other or by other parts of the game.

Sensors - The sensors have to plug-in into the platform system event system and the game engine to obtain information about the world. The fact that the agent relies on events generated by the system hence requires the system to send events for the agent to utilize.
Controller - GOAP and SHOP can be used interchangeably. For the current implementation of the game, the use of only GOAP should, however, be sufficient. The controller can be defined as an abstracted interface for GOAP, SHOP and possibly other controllers. If the game would be extended in the future with storytelling, SHOP could easily be added as a controller for storytelling characters. GOAP based controllers can be computationally heavy with a large database of actions, hence the platform needs to support computationally heavy search tasks while still providing smooth rendering.
Managers - Multiple managers can be implemented individually as modules of the agent. Managers can co-exist if they are disjoint, but if they need to communicate an interface is needed, possibly resulting in dependencies.

Blackboard - The dependencies among managers can be solved by means of a blackboard. A blackboard also promotes an extendible architecture as it decreases direct dependencies among modules. A blackboard, however, constrains the kind of information that is exchanged between modules, as only information supported by the blackboard can be exchanged.

Architecture



The synthesized architecture of an AI agent is shown in the figure above. Events generated by the platform are processed in the sensor and used to update the world model. The world model is a database of various facts. The variability in the facts that can be stored is handled by the strategy pattern. The actions that the AI takes are determined by the AI controller, for the current specifications only the GOAP controller is implemented.

The GOAP controller generates the goals and actions of an NPC on runtime. The goal manager selects a goal given the current world model and then proposes a plan with actions to obtain that goal. The AI module executes these actions, by delegating the current action to be executed to the managers and by controlling the actual NPC. The NPC is a game unit. The NPC will consume primitive actions derived from the actions of the controller and actions sent from the managers, such as looking, aiming and moving, and communicate them as

input to the platform (as events). The managers perform more fine-grained actions, such as walking the right path, managing health, and using the weapons. The managers can operate concurrently, and they communicate using the blackboard. For example, if the action is to shoot a certain target, the WeaponsManager can retrieve the target from the blackboard, find the best weapon to use and then generate concrete operations, such as shooting with weapon Z in direction Y, which the AI module passes to the NPC.

Variability management

The variability in goals and actions for NPCs is handled by defining a language in which these actions can be defined, which is then interpreted at runtime. This allows the introduction of agents with different goals and a different set of actions as new levels and scenarios are added.

The variability of the controller is managed by abstracting all controllers to use a single interface, hence allowing the implementation of the controller to be dynamically changed as needed, making it possible to switch between GOAP and SHOP if this is needed in the future. By abstracting the controller, not only the actions and goals of the agents can easily be adapted, but also the way that actions are selected to obtain these goals, resulting in a great adaptability of NPC behaviours.

The concept of managers is introduced to manage more complex behaviours such as team behaviour and graceful degradation. To maintain multiple managers and to facilitate scaling the number of managers, a blackboard is proposed to support manager communications. A blackboard removes direct dependencies between managers and facilitates efficient and adaptable communication between managers. If the requirements change, and the current managers need to be changed or extended, this can be done easier without breaking cross-manager dependencies. For example, if the graceful degradation manager is not included in the first version of the game, the agent can easily be adapted to include such a manager.

References:

Russell, Stuart J.; Norvig, Peter (2003), Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-790395-2, chpt. 2

Long, E. (2007), Enhanced NPC Behaviour using Goal Oriented Action Planning, University of Abertay Dundee, Division of Software Engineering. *http://www.edmundlong.com/downloads/Masters_EnhancedBehaviourGOAP_EddieLong.pdf*

Dicken, L. (2012), Game AI 101 - NPCs and Agents and Algorithms... Oh My! Bradley University, Peoria, IL. https://www.slideshare.net/LukeDicken/game-ai101key

Vassos S. (2012), Introduction to AI Strips Planning, University of Athens, Greece. https://www.dis.uniroma1.it/~degiacom/didattica/dottorato-stavros-vassos/AI&Games-Lecture1-part2.pdf

Miyashita, S., Lian, X., Zeng, X., Matsubara, T., & Uehara, K. (2017). Developing game AI agent behaving like human by mixing reinforcement learning and supervised learning. In 2017 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). IEEE. https://doi.org/10.1109/snpd.2017.8022767

Cristea, V., Pop, F., Dobre, C., & Costan, A. (2011). Distributed Architectures for Event-Based Systems. Studies in Computational Intelligence, 11–45. https://doi.org/10.1007/978-3-642-19724-6_2

Horswill, I., & Zubek, R. (1999). Robot Architectures for Believable Game Agents. In Proceedings of the 1999 AAAI Spring Symposium on Artificial Intelligence and Computer Games

Miyake, Y. (2016). A Multilayered Model for Artificial Intelligence of Game Characters as Agent Architecture. Mathematics for Industry, 57–60. doi:10.1007/978-981-10-1076-7_7

Corkill, D. P. (October 2003). Collaborating Software: Blackboard and Multi-Agent Systems & the Future. Proceedings of the International Lisp Conference. New York, New York

Solution Domain: Game Units

By Tycho Braams

Solution domain description

The game will be played by a player interacting with game objects, or game units, in the game world. For a game to function, many different (types of) game units are required. A solution needs to be found for the problem of supporting and configuring these game units. Game units could be defined individually and thus hardcoded. However, this will make it necessary to define all units from scratch, for every unit that is required without a possibility of reuse. Since the game will require a lot of game units, across a multitude of game levels, this will lead to a lot of work.
Furthermore, it should be possible to adapt or extend the units in the game. Hardcoding the game units is not a sustainable solution for this problem. Some kind of structure needs to be found that describes how game units can be defined.

Structure/Pattern: The first thing that comes to mind when discussing a structure for objects that share (some) properties is the use of inheritance. The problem with this approach is that game units that are very similar, but still different, will be different instances. This will lead to an unmanageable number of subclasses[5].
Another possibility is defining a language to describe the objects using the interpreter pattern [6]. A grammar can be used to define what configurations of game units are allowed and what combinations are not allowed. This requires the people working with game units to understand the language, but if a combination is invalid, the game engine can detect this using the grammar. Simple changes such as new units can be defined in the language, while other more complex changes would only require a change in the language and grammar, leaving the rest of the architecture as it is.
The most common pattern in the gaming domain is the use of Composition. Components are defined that are responsible for specific attributes, for example, graphics or physics.
A game unit then becomes a composition of these different components.[5][7][8]
This can be seen as a simple version of a language with an implicit grammar.
Unity is a popular game engine that is based heavily on the use of GameObjects with Components.[9]
Since the game units need to be extendable and extra objects can be added due to extra payments, components should also be extendable. A good way to do this is by using the decorator pattern with objects that are likely to receive extra functionality.[10] This also allows game units to be extended at runtime.

Editor: An editor provides an interface to the people working with the game to create and configure game units or components. Some editors serve a specific purpose, such as creating a game stage/level. Other editors can create entire games. If a language is used to define the game units, the editor can provide a more intuitive view to the developers, reducing the required knowledge of the developers.

Graphical detail: A common technique used with graphics to manage the number of details that are rendered for a game unit is Level of Detail.[11][12] This technique renders less complex versions of units that are further away from the viewpoint of the player. This decreases the workload of the renderer.
This technique can also be used to adapt the level of detail to the hardware of the player. The units in the solution should support the storage of multiple graphics versions with different complexities.

Animation: Animation is a very complex process. To ease this process, models are often given a skeleton that can be used to lead the movements in the animation.[13] The structure of the skeleton can also be used to map animations from one character to another. In a war game, players and non-players alike all use humanoid forms. By including a skeleton in the unit, animations only need to be created once and can then be mapped to all the humanoids in the game.
Such a skeleton can also be used to customise a character or to help in the use of complex physics.

Current developments and future trends

Automation
The complexity of games and the worlds that they are played in is growing quickly. It is becoming harder to make the game worlds realistic while still providing a size that players have become accustomed to. Research is being done to figure out how to improve the reality of the game world without massively increasing the costs and workload.[14][15] Researchers are also working on the generation of game worlds that adapt to the player.[16]

Reuse
Since the gaming industry has become a large economy and games are developed with increasing budgets, it has become very lucrative to make game components reusable. The current trend is to split the architecture into components that are very game-specific, which are unlikely to be reusable and components that are less game-specific. Such components can then be used in the creation of new games, decreasing the overall cost when developing multiple games.[17]
This also makes it possible to buy or outsource certain parts of the game. Many components in game development require very specific domain knowledge. For a company to develop all the components themselves, the developers will need expertise in all of these domains.[18]

Commonality & Variability analysis

Commonalities
Game units will always need a way to indicate their position in the game world. This is also strongly linked to the size or scaling of the object. If game units are visible, the game engine needs to know how to render the model. As such, the game units should include information about their graphical representation. While looking at the solution domain, solutions that also support adaptability and extensibility of units, attributes that are always needed are stored in the main object, while other attributes are stored in separate components.

Within the war game genre, always present elements are characters, weapons, vehicles, buildings and terrain. The game also needs less intuitive objects such as lighting sources and cameras for the view of the player.
To regulate the behaviour of the objects in a game, objects need to respond to the physics defined by the game engine.

Variabilities
For the game units in a game, the first type of variability is the type of units that are present in the game. Some elements are very simple and only require a position and a graphical representation. Examples of such objects are trees or rocks, static objects that are there to shape the game world. Other elements are far more complex, for example, a player character. Such a unit needs a position and graphical representation, but it also needs a way to move in the game world, interact with the game world and in this game, shoot at enemies.
Another variability comes from the graphical representation. There are multiple tools that can be used to create graphical models and textures, each with their own data format. Examples of these tools are Blender, Cinema4D, Rhino and Autodesks's 3Ds Max, Maya and AutoCAD.[19]
Furthermore, a variability is introduced by the varying quality of hardware that needs to be supported by the game. When the game is run on a very new, high-quality system, the game units can and should have a graphical representation with a lot of detail. Because the game should also run on systems with a lower quality, the level of detail should be scaled to the quality of the system that is used to run the game.

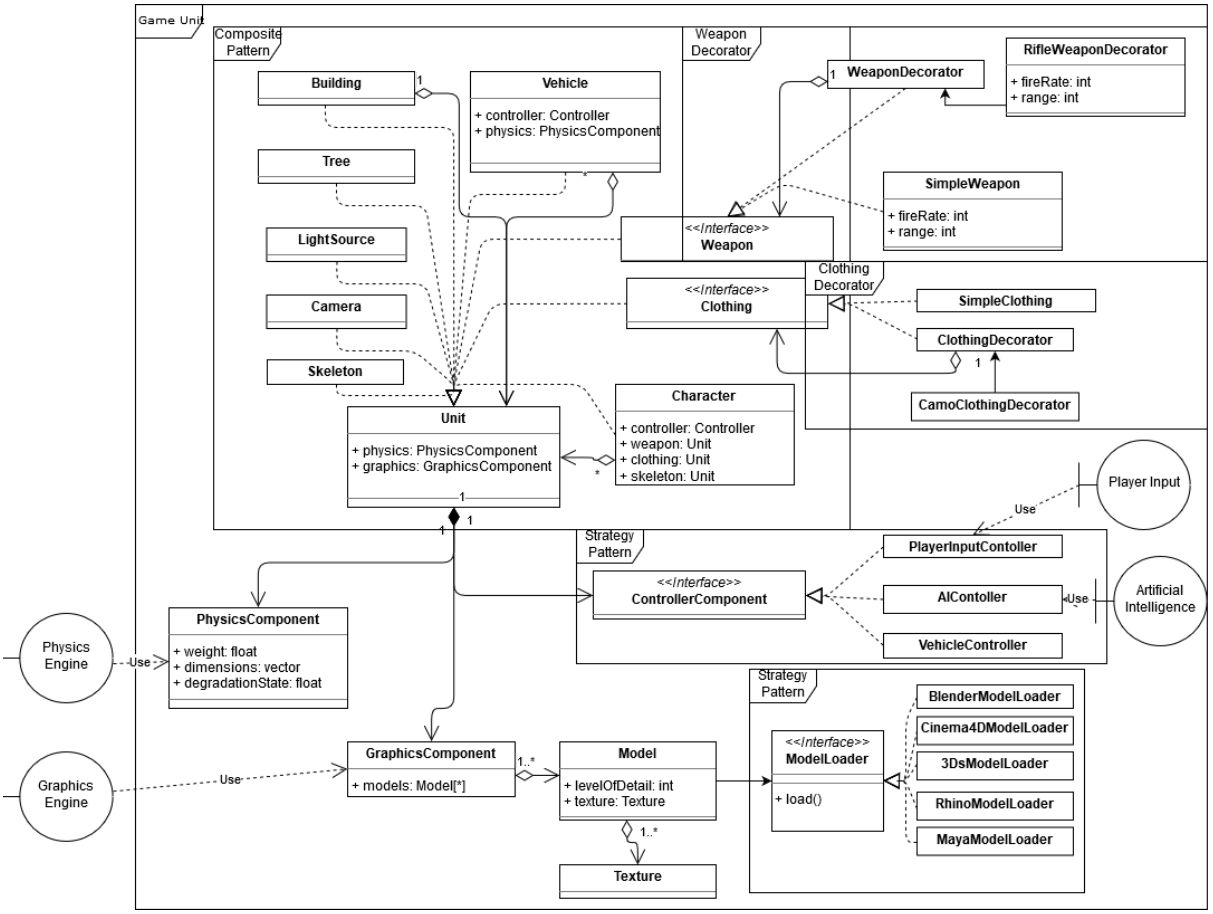
Constraints

The chosen solution for the game units has an influence on the architecture of Artificial intelligence and the architecture of the game engine. The artificial intelligence needs to control a game unit, namely an enemy character. How the game unit stores information about for example the weapon of the game unit influences how the artificial intelligence can use this information.
If a grammar is defined for the game units that restricts the possible combinations of elements, the game engine needs to be able to check and enforce the rules described by this grammar.
Finally, the selection of components will heavily impact the structure of the game units that are used in the game. The game developers will likely try to adapt the predefined structure to their needs. As such, the components should be chosen in such a way that extra components might be created if needed, but that should not be necessary. Furthermore, all the component should have a clear use and responsibility.

Architecture

In order to reduce the interdependencies in the total architecture, it would be better not to rely on other solution domains to verify the validity of defined game units. This also makes it easier to reuse the architecture for this solution domains when designing new games. It was therefore decided to use a composition of components to model the game units.
From the commonality analysis, a number of components that should be included have become clear.

These are a graphics component, a physics component and a controller component. A composite pattern is used to implement the different game unit types. For example, a game unit can be a character who is wearing some clothes and holding a weapon.
This allows game units to adaptable and extendable.
As weapons and clothing can be changed by in-game purchases, the decorator pattern is used to make these objects very flexible. A single decorator is shown in the architecture, more can be created by the game developers whenever such decorators are required.
The graphics component can contain a set of models, which can be used to render the game unit at different levels of detail.
A Strategy pattern is used to manage the model loaders for the different types of graphical modelling tools.
The same pattern is used to manage how game units such as characters and vehicles are controlled.
The choices of controllers, model loaders and game unit types are only meant as a starting point. The game developers can use an editor to create new versions if they are required.



Variability management

The patterns applied in the architecture are meant to manage the variabilities that were identified in the variability and commonality analysis.

The strategy pattern is meant to deal with the variability introduced by the different tools for graphical modelling. It also makes it possible to add support for other model types if developers are added that have preferences for other tools
The composite pattern is used to allow the developers to customize the game units to their needs.
The decorator pattern is used to manage the unknown changes that will likely be made to the weapons and clothing in the game.
Some components require information about other components to be able to function. To make sure these links are performed correctly, dependency injection should be used. This prevents problems with null pointers that could occur if the developer has to set the dependencies manually.
Future changes involving degradation are managed by maintaining the degradation state of a unit in the physics component. This can then be used by the game engine to render a different model that shows some amount of damage to the unit or to change the performance of the unit.

References:

[5] Passos, E. B., Sousa, J. W. S., Clua, E. W. G., Montenegro, A., & Murta, L. (2009). Smart composition of game objects using dependency injection. Computers in Entertainment, 7(4), 1. https://doi.org/10.1145/1658866.1658872
[6] Buro, M., & Furtak, T. (2005, August). On the development of a free RTS game engine. In *GameOn'NA Conference* (pp. 23-27).
[7] Nystrom, R. (n.d.). Component. Retrieved November 8, 2018, from http://gameprogrammingpatterns.com/component.html
[8] Entity–component–system. (2018, October 19). Retrieved from https://en.wikipedia.org/wiki/Entity–component–system
[9] Unity Technologies. (n.d.). GameObjects. Retrieved November 8, 2018, from https://docs.unity3d.com/Manual/GameObjects.html
[10] Brkusanin, P. (2013, December 9). Decorator pattern usage on a simplified Pacman game. Retrieved November 8, 2018, from https://www.codeproject.com/Articles/690410/Decorator-pattern-usage-on-a-simplified-Pacman-gam
[11] Giang, T., Mooney, R., Peters, C., & O'SULLIVAN, C. A. (2000). ALOHA: Adaptive Level of Detail for Human Animation: Towards and new Framework.
[12] Strothotte, T., Cohen, J. D., Reddy, M., & Schlechtweg, S. (2002). *Level of Detail for 3D Graphics*. Morgan Kaufmann.
[13] Skeletal animation. (2018, August 21). Retrieved November 8, 2018, from https://en.wikipedia.org/wiki/Skeletal_animation
[14] Kessing, J., Tutenel, T., & Bidarra, R. (2012). Designing Semantic Game Worlds. In Proceedings of The third workshop on Procedural Content Generation in Games - PCG'12. ACM Press. https://doi.org/10.1145/2538528.2538530
[15] Kessing, J., Tutenel, T., & Bidarra, R. (2009). Services in Game Worlds: A Semantic Approach to Improve Object Interaction. In Lecture Notes in Computer Science (pp. 276–281). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-04052-8_33

[16] Lopes, R., Eisemann, E., & Bidarra, R. (2018). Authoring Adaptive Game World Generation. IEEE Transactions on Games, 10(1), 42–55. https://doi.org/10.1109/tciaig.2017.2678759
[17] Folmer, E. (n.d.). Component Based Game Development – A Solution to Escalating Costs and Expanding Deadlines? In Component-Based Software Engineering (pp. 66–73). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-73551-9_5
[18] Plummer, J. (2004). *A flexible and expandable architecture for computer games* (Doctoral dissertation, Arizona State University).
[19] Creger, R. (2018, August 24). 8 awesome options for 3D modeling software. Retrieved November 8, 2018, from https://99designs.nl/blog/design-resources/3d-modeling-software-guide/

Solution domain: Platforms

By Jan Boerman

Solution domain description

The platforms solution domain provides many possible solutions to the game server hosting side of things. There are multiple possible solutions, such as a peer to peer (p2p) architecture, or a classic client/server architecture. Player hosted servers are also an option. The solution domain also includes non-game servers such as the database servers, matchmaking server, asset servers and transaction services. In this section, we discuss networking protocols, and other decisions having to do with networked servers. We will quickly touch upon security, but will not go into detail.

Current developments and future trends

Layered Architecture
The layered architecture is a traditional way of structuring programs. A very generic program has a Presentation Layer, a Business Layer, a Persistence layer, and a database. This structure is often perceived as a natural fit for standalone applications. The user interacts with the presentation layer, which in turn interact with the business layer, which in turn interact with underlying layers, etc. The ‘interactions’ can be of any kind - in memory subroutine invocation, sending data over the network, interacting with the file system, sort of interaction is allowed.

Event-Driven Architecture
The event-driven architecture is a flavour of a layered architecture, that represents the interactions as events. Events can be generated by one layer and received by others. The difference is an inversion of control: The receiver of events decides which events to listen to, instead of the event poster deciding which receivers receive the event. A rule of thumb is that different layers don’t produce the same event

Cloud Architecture
Currently, many wargames make use of a client/server architecture. The client being a programme on the end user’s machine, and the server being a programme that performs game logic and is hosted by the game provider. Examples of such games are: League of Legends, Battlefield, Dota2, Call of Duty.
More recently cloud architectures - a flavour of the client/server model - gained in popularity. Cloud architectures have one big advantage, they scale very well horizontally. That is, if the game sessions are reaching their maximum capacity of players, the architecture spins up new nodes that players can connect to. The reverse is also true, if game server nodes are used by just a handful of players, then it can be turned off, saving on cloud resources. A popular game that uses this architecture is Fortnite.
There is also the possibility to ‘host’ one part of the game world on one node and another part of the game world on another, such that players in the same world are connected to different servers - decreasing the load on a single node.

Authentication
Since the game will feature purchases, users have an account that ‘own’ the purchased items. This is common for many modern multiplayer games, and to make sure nobody else can access the purchased items, players log in to the service.
There are many choices for authentication protocols, but they fall into two categories:

Point to Point protocols (PPP)
This category is generally suitable when servers need to validate the client’s identity before the client can access data on the server. Typically authentication works through username/password authentication or challenge/response authentication.

Authentication, Authorization, Accounting (AAA)
These types of protocols are more complex and used for larger networks. Authentication is still typically done through username/password authentication.

Commonality & variability analysis (platforms)

Commonalities
For the analysed architecture styles immediately one thing becomes clear: They don’t rule out each others’ choices. Event-driven architectures can be layered, and cloud architectures can host software in layers. For the layers it doesn’t matter how the events are exchanged between the layers, that can be abstracted away in the program code using design patterns. The most obvious commonality is that the architectures used in most modern war games are client/server architectures. Peer2peer is less used since servers can do verification of incoming network packets, and it doesn’t scale well as the number of clients increases.

Variabilities
The type of protocols used for communication of events or messages between layers is not fixed. Any protocol that works for other games should work for ours. There are multiple degrees of cloud architectures too, one can use a clustered architecture where the game clients communicate to game servers, but the servers also communicate with each other. Through a distributed protocol they may ‘decide’ to bring a new node up or down. The other end of the spectrum is a full-on horizontal scaling solution where an external controller manages how many game server nodes are present. While both options can be used when the game world is split up over multiple game nodes, we think the first option fits best since a cluster can then host one world.

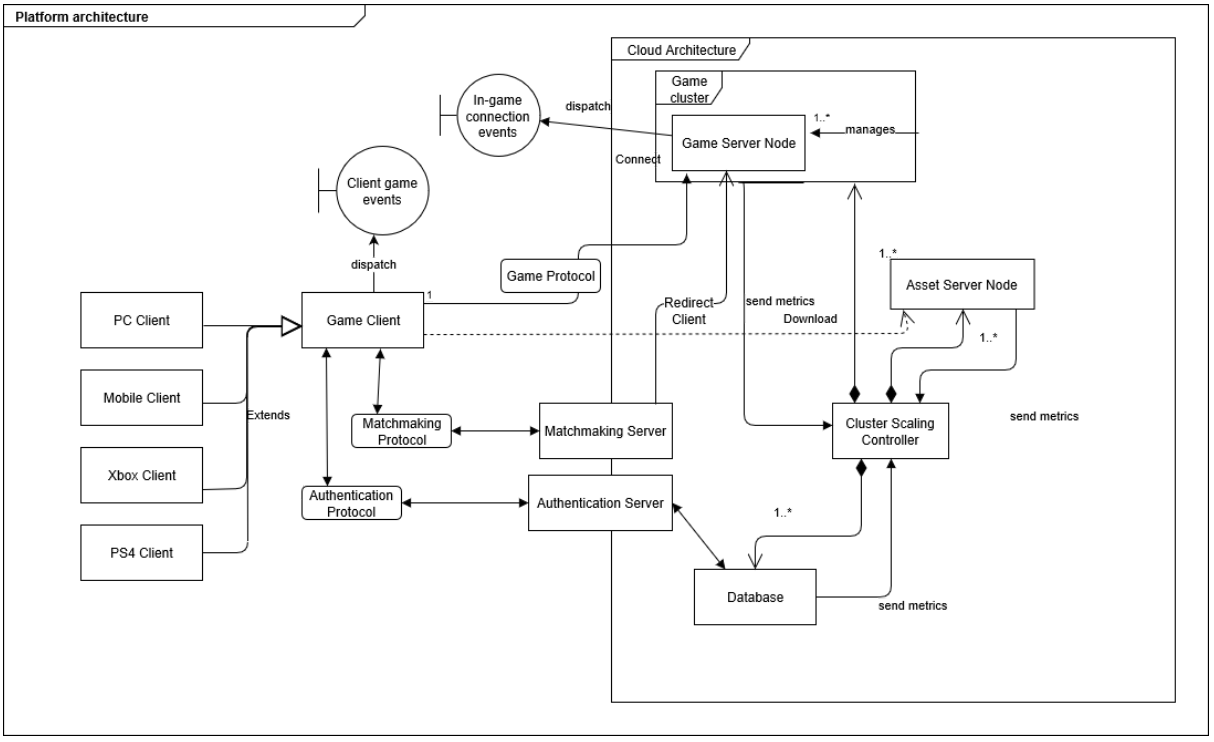
Commonality & variability analysis (authentication)

The commonality between AAA and PPP authentication protocols is that both allow username/password-based authentication. We may ‘remember’ a token or hash value on the client so that users don’t need to log in every time they start the game.

Constraints

There is still a theoretical limit as to how many (clusters of) game nodes can be managed by a controller server. The number of servers in a cluster can also not grow too large, or there will be congestion in the communication. We, however, don't think this is a problem with current cloud computing power.

Architecture



The platform architecture looks as follows, there are multiple types of clients as required by the cross-platform requirement. They are not all drawn, but they are all 'Game Clients'. The game client is the programme that runs on the users' hardware and performs the task of authenticating the users. There are some authentication servers in the cloud that the clients will try to connect to, and a fixed authentication protocol is used.
After the users are 'logged in', they can start playing games. This is done by connecting to matchmaking servers. The programmes running there will try to find players that are 'close' to each other physically to form a group. Once the group is complete, the matchmaking server kicks off a game.
When players enter a match, their clients connect to a Game Server Node in a cluster of game servers. These clusters can host one giant game world each, and the clusters are managed by a cluster scaling controller. The game server nodes generate the connection events that are used in the game engine.
All clusters are in a feedback loop with at least one cluster scaling controller so that the controller can decide whether new game node clusters must be created in the cloud.
The asset servers are there so that clients can download sounds, models and textures of in-game units. Their number is also scaled up and down by a cluster controller server.

Lastly, there is the database in which all relevant game data, as well as users credentials, are stored. Since the database is part of the cloud, there can be multiple nodes e.g. a master/slave system. This hasn't been explored in detail.

Variability management

Matchmaking
The number of players connected to one cluster can vary a lot, and there may be different matchmaking algorithms that work best for certain numbers of players. We opt to use the strategy pattern here. The optimal matchmaking strategy per player count can be learned beforehand through simulation and runtime monitoring.

Protocols
The game's protocols are not fixed by the platform architecture, but we manage this variability through containment. That means that a protocol that works can be picked and then we stick to it.

Game nodes
The number of game nodes is managed by a cluster, which is managed by a game Cluster Scaling Controller. The number of game nodes is defined through a compositor pattern. The more players per game session, the more game nodes in a cluster are needed, and the more game sessions, the more clusters are needed.

References

[20] Hsu, C. A., Ling, J., Li, Q., & Kuo, C.-C. J. (2003). The design of multiplayer online video game systems. In A. G. Tescher, B. Vasudev, V. M. Bove, Jr., & A. Divakaran (Eds.), Multimedia Systems and Applications VI. SPIE. https://doi.org/10.1117/12.512201
[21] Moraal, M. (2006). Massive multiplayer online game architectures. *Whitepaper, January*.
[22] Xu, Y., Nawaz, S., & Mak, R. H. (2014). A Comparison of Architectures in Massive Multiplayer Online Games.
[23] Bharambe, A. R., Pang, J., & Seshan, S. (2006, May). Colyseus: A Distributed Architecture for Online Multiplayer Games. In *NSDI* (Vol. 6, pp. 12-12).
[24] Woo, T. Y. C., & Lam, S. S. (n.d.). A semantic model for authentication protocols. In Proceedings 1993 IEEE Computer Society Symposium on Research in Security and Privacy. IEEE Comput. Soc. Press. https://doi.org/10.1109/risp.1993.287633
[25] Lloyd, B., & Simpson, W. (1992). *PPP authentication protocols* (No. RFC 1334).
[26] Richards, M. (2015). Software Architecture Patterns. Retrieved November 11, 2018, from https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/
[27] Dyl, C., (2018), Director of Platform, Epic Games url: https://www.youtube.com/watch?v=HjQyTHXOTeI

Solution Domain: Game Engine

By Marko Groffen

Brief Description

A game engine is a development environment that can be used to create (features of) computer games. Game engines can include multiple core features, for example, the rendering of 2D/3D graphics, a physics engine, sound, animation, AI, scripting and networking. These features can be used to solve the concerns about the user interface, the memory and visual layout for different platforms, the physics of the game world and the graceful degradation of objects. The game engine also provides a runtime environment for the game, which will mean it should support the solutions for the game units and the artificial intelligence since the runtime environment also simulates those features. It is common for game engines to be reused for multiple games to make the creation of one economically feasible, as well as the fact that most engines support multiple gaming platforms, making the deployment of a game on multiple platforms easier for the developers. This gives a large variety of game engines that are already created by other development teams, which can potentially be used for our game. [28] [29] [30]

Physics Engine:
An important part of the realism is created by the physics engine in a game, which provides a description for location and velocity calculations as well as an interpreter of collisions. It also supports physical behaviour in the form of, for example, a rigid body, soft body and fluid dynamics. Most of the shelf game engines have a physics engine build into them, often giving different solutions for 2D and 3D games. Our focus only lies on the 3D physics engines, however, since our game will be developed in 3D. The degradation of objects within the game can also be visualized with the usage of the physics engine. [31] [32]

Current Developments

The first game engines started popping up in the 90s when advances in computer hardware allowed for more complex features in games. The introduction of features like 3D graphics and realistic physics engines were implemented in the game engines with the idea of reducing development time for games since engines could be reused for later projects. It also gave a high-level language solution for scripting, leaving the more complex calculations in a low-level programming language to the game engine. The usage of a high-level programming language came from the idea to increase development speed instead of giving the optimal performance.
Since a large part of the platforms our engine should support is relatively new, the amount of engines that support all of these platforms is limited. Older existing game engines are often not updated for newer platforms. Since the game should be easily deployable over the various platforms, using a game engine that supports them all is key. Working with various engines makes the cross-platform integration more difficult due to different game architectures. The 2 leading game engines in the current market, which also support all of the required platforms for our game, are Unreal Engine and Unity. [33] Unreal Engine is

mainly used for games run on the more powerful platforms, like PC, PlayStation and XBOX, and is used by large game developers in the likes of Ubisoft and Activision. [34] The Unity engine is more popular with free-to-play games and mobile devices, but has nonetheless customers in the likes of Electronic Arts and Square Enix. [35] Unreal Engine uses C++ for its scripting, while Unity uses C# and Javascript for scripting and C++ during runtime.

While these engines are used by large companies, most triple-A game developers (games with a multimillion-dollar budget) create their own engine for a specific game. This gives them more influence over the capabilities of the engine and does make them independent of third-party software. Depending on the budget of our game, since the development of a game-specific engine is expensive, this can also be a viable alternative to the previously described Unreal Engine or Unity. So this leaves us with 3 options:

- Unreal Engine by Epic Games
- Unity by Unity Technologies
- A game specific engine

A commonality and variability analysis will be performed between the Unreal Engine and Unity. Based on this analysis one of the 2 engines will be chosen if they prove to be sufficient for our game. If this is not the case, a game specific engine will be introduced making up for the lacking specifications of the engines.

Commonality and Variability Analysis

Both engines have a large commonality in the way the engine is divided into sub-functionalities. Both use a graphics API, a third party physics engine, game units, AI tools, scripting tools and tools to create a user interface. These functionalities are sufficient to give a solution to the specified concerns. The implementation of these features is, however, done differently and is discussed below [36] [37].

Graphics API
For graphics and lighting options, both engines make use of a graphics API (application programming interface). They both make use of the DirectX, within specific the Direct3D feature for 3D level design, developed by Microsoft. This graphics API is widely used for game design, especially for the Windows and Xbox platform, since these platforms are also developed by Microsoft. The usage of a standard graphics API makes it possible for the designed lighting and graphics to be adapted to various execution platforms.

Physics Engine
As for a physics engine, both game engines make use of the PhysX engine developed by Geforce Nvidia. This makes the possible physics engine features very similar, however, the way they are implemented within the tool are different. Unity makes a separation between differently shaped colliders, joints of objects and rigid bodies. They also define specific ragdoll physics for character objects. Unreal engine defines collisions based on the type of object, for example, a vehicle or a body, and does not use joints. Both allow the design of specific clothing physics. In addition, Unreal Engine has the ability to specify the fact that an

object is destructible and gives, therefore, options about the way the object can be destroyed. In unity, the destruction of objects has to be manually applied by dividing the object into separate shards that can fall apart. Since degradation of objects is an important requirement, Unreal Engine gives the easier solution here.

Game Units
Both engines allow for the creation and the importing of game units. Both engines also have a direct access to a large library of game units, created by the engine developers or third-party developers. This means both engines give the option of quick level design by providing a large number of asset resources. Between the two engines, Unity currently has the larger game units library of the two.

Scripting
For scripting, there is a significant difference between the implementation of the two engines. Unreal Engine uses C++ for scripting in combination with a third party compiler of choice (Visual Studio for example). Unreal Engine also includes a visual scripter called Blueprint. It can be used to create gameplay elements with the use of a node-based system. The scripting system is used to define object-oriented classes and objects, and can therefore also be extended with C++ code. Unity, on the other hand, uses C# or JavaScript as the scripting language and uses Mono as a scripting framework for C#, which includes a compiler, a runtime and various framework and Mono specific libraries. Unity does not have a visual scripter in the likes of Blueprint.

Artificial Intelligence
To create Artificial Intelligence for the in-game NPCs, both engines implement different approaches. Unreal Engine makes use of a blackboard in combination with a behaviour tree. The blackboard functions as the AI's memory and contains the key values for the behaviour tree. The behaviour tree is the processor of the AI, where decisions based on environmental input are made and where these decisions are then acted upon. Unity, on the other hand, does not make use of a blackboard or behaviour tree. Instead, the AI's are completely event-based and make use of a NavMesh, a mesh within the game world to control and direct AI navigation throughout the game. This means it is more difficult to create sophisticated AI models in Unity due to the AI implementation being more constrained.

User Interface
To create an in-game user interface, like a menu screen or Head-Up Display (HUD) for the player within the game, Unreal Engine has a dedicated tool. The Unreal Motion Graphics UI Designer (UMG) is a visual UI authoring tool that consists of predefined widgets, like buttons, sliders or progress bars. This makes the creation of a user interface relatively simple. Unity, on the other hand, does not provide such a specific tool. It has the OnGUI function within the scripting API to provide rendering and the handling of GUI events, but the UI in its completion has to be designed using scripting.

Constraints

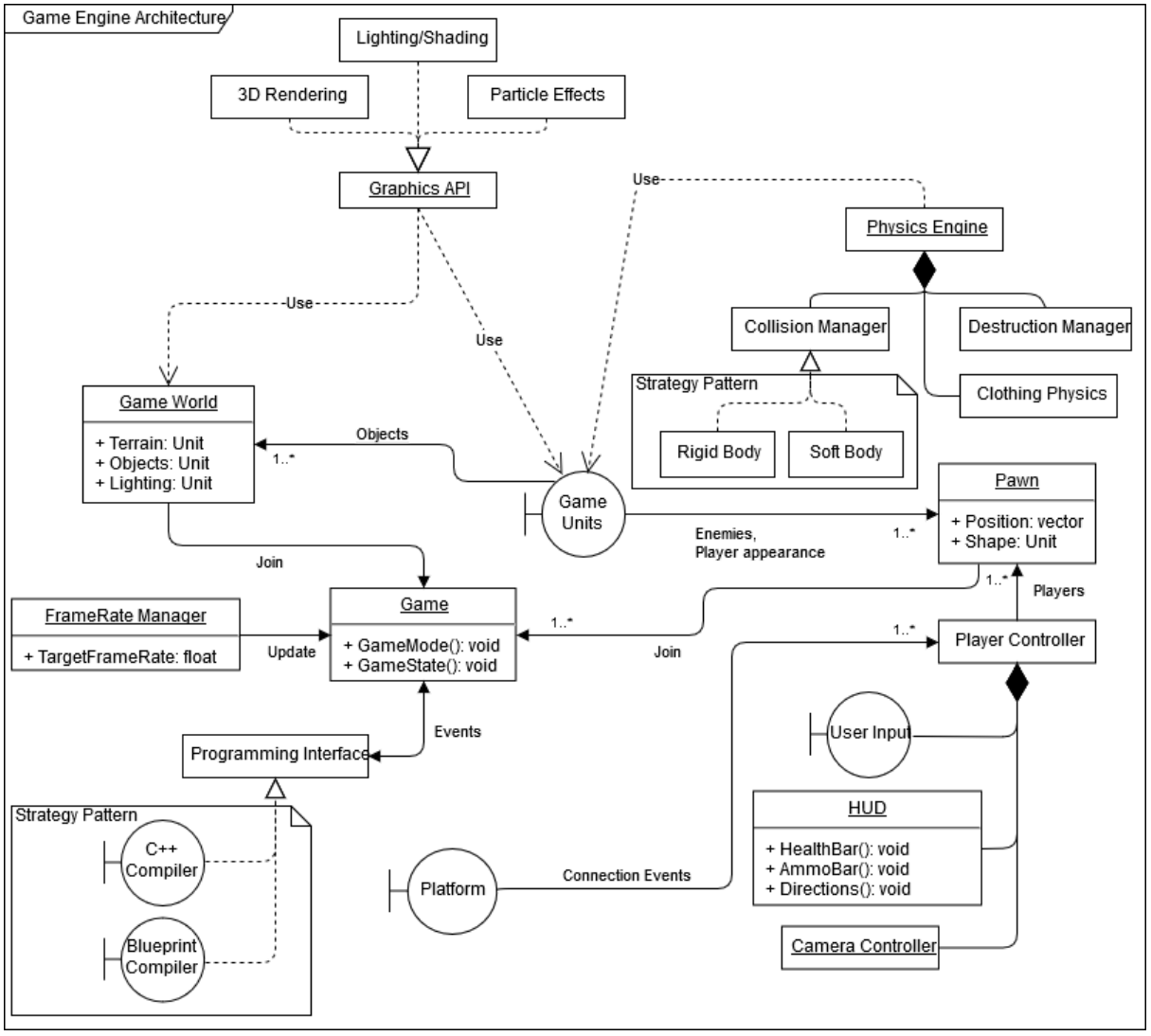
The chosen game engine will imply constraints on the chosen Artificial Intelligence, Game Units and Platform solutions since the game engine is the linking and combining factor between these architectures. The choice of Artificial Intelligence implementation within the game engine has a large influence whether the usage of a blackboard is supported or not. Whether the AI can operate goal-based or event-based is also influenced, with Unreal Engine supporting both.
Both engines allow importing object designs for game units, but the allowed format is limited, constraining the possible tools allowed to design the game units.
The scripting implementation handles the events within the game. Since the game is played online, part of these events are influenced by inputs from the platform, making a certain programming language a potential constraint in handling such events. Unity is also constrained to just traditional scripting, while Unreal Engine gives more freedom with the addition of Blueprint, which can be used in parallel with C++. The same holds for the design of the UI; where Unreal Engine has a dedicated visual design tool, Unity is constrained to a scripting solution.
The implementation of destructible objects within the game is mostly constrained when Unity is chosen as the game engine since Unreal Engine has dedicated object design features for destructibility. Unity is limited to the creation of shard objects for game units to represent destruction, limiting the realism of the destruction.
The lighting and rendering are constrained in the same way for both engines since they both use the same third party graphics API: DirectX.

Both Engines do support access to the source code of the engine, which gives a larger control over all the features of the entire engine. This can reduce the constraints of the engines since the described features can be changed or even replaced. In this case, however, a large knowledge of the source code is needed as well as more development time for the game.

Architecture

Based on the commonality and variability analysis and the constraints, it is chosen to use the Unreal Engine as the base for the game engine architecture. It gives fewer constraints and has features that will directly contribute to the solution of the concerns, like the destructibility specifications for the physics engine or the UMG tool for the UI. Also, the way artificial intelligence is implemented in Unreal Engine is similar to the solution represented in the AI solution domain, using a blackboard. Unity requires a complete redesign of the AI feature.
Most of the features of the engine can directly be represented within the architecture.
For the physics handling, the choice of soft or rigid body collision is managed by the collision manager and is decided during runtime. The collisions, clothing physics and destructibility of objects are part of the physics engine.
The graphics API implements rendering, lighting and particle effects, to be used within the game world or by the game units.
The player controller, of which multiple exist since it’s a multiplayer game, contains the user input from the physical controller, connection events from the platform, the HUD and the

camera controller. Both players and NPC are represented as Pawns within Unreal Engine, which are controlled by the player or the AI through the game units respectively.
The game world, which takes objects from the game units is combined with the pawns into the game, which contains the game mode (e.g. game rules and win conditions) as well as the game state (e.g. connected players, score, object locations, progress within a level).
A frame-rate manager makes sure the game is constantly updated. The game events are handled by the programming interface, that interprets either the C++ compiled scripts or Blueprint compiled scripts.



Variability Management

In order to manage the variability in user input due to the different gaming platforms, the user input is modelled as a boundary object, meaning the input is handled by the platform and not by the game engine. It only interprets the given input. Also, the HUD and camera controller are defined as separate objects and can be swapped around based on the dedicated platform. This means a change in platform does not influence the actual game or events.

Whether the C++ or Blueprint scripting method is used, both will be recognized by the programming interface. The strategy pattern represents the fact that both will be interpreted at runtime and are interchangeable.

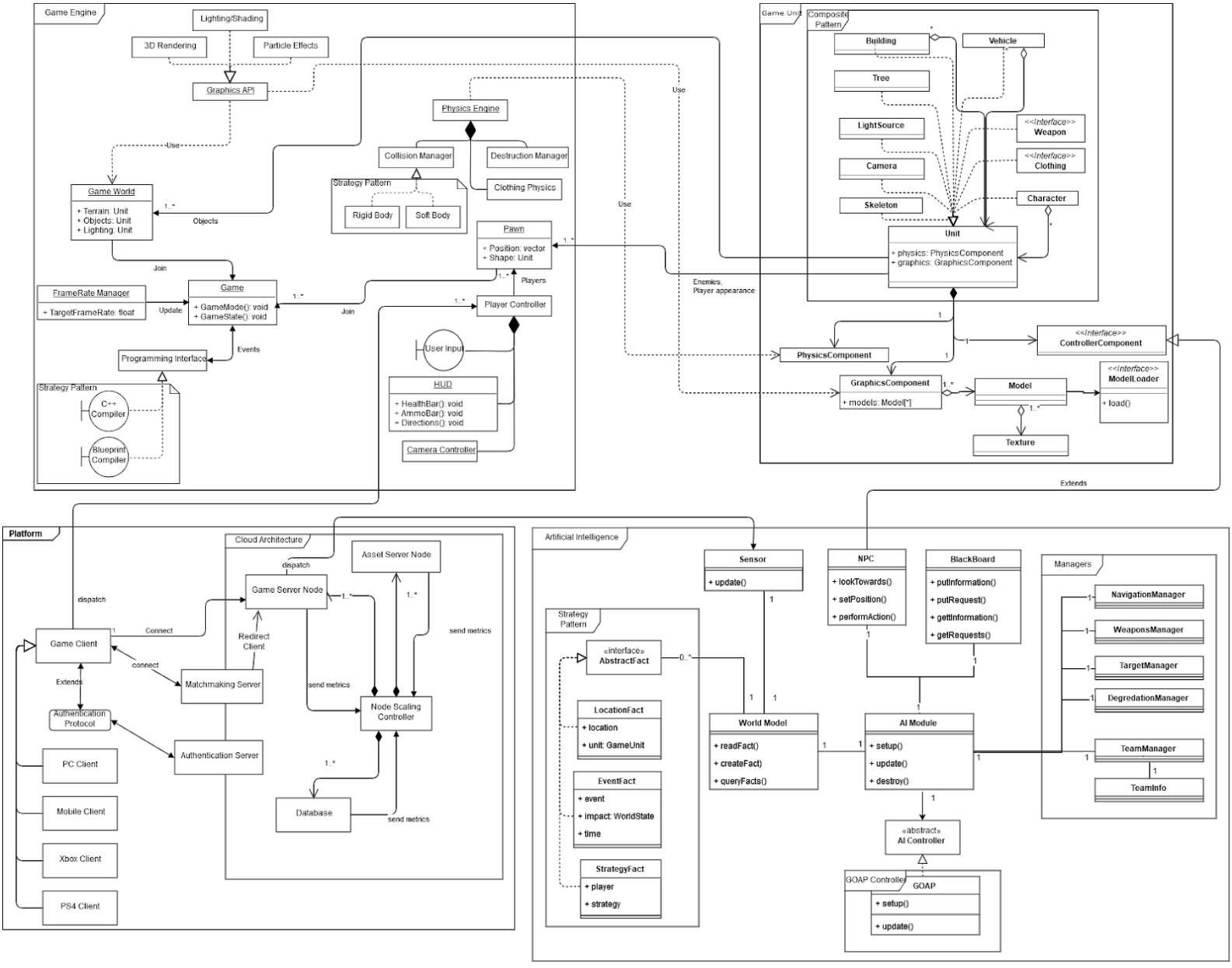
Both the graphics API and Physics Engine are modelled as separate objects and can be changed when the source code of the engine is altered for improvements. The only requirement is that the features represented in the architecture are still implemented since both the game world and game units are depending on these features.

References:

[28] Gregory, J. Game Engine Architecture. 2009 https://www.gameenginebook.com/
[29] Nilson, B. and Söderberg, M. 2007. Game Engine Architecture. Mälardalen University. http://citeseerx.ist.psu.edu/viewdoc/versions?doi=10.1.1.459.9537
[30] Caltagirone, S. Keys, M. Schlief, B. Willshire, M.J. (2002) Architecture for a massively multiplayer online role playing game engine. Journal of Computing Sciences in Colleges, Volume 18 Issue 2, December 2002, 105-116 https://dl.acm.org/citation.cfm?id=771339
[31] Choi, J., Shin, D., & Shin, D. (2006). Research on Design and Implementation of Adaptive Physics Game Agent for 3D Physics Game. In Agent Computing and Multi-Agent Systems (pp. 614–619). Springer Berlin Heidelberg. https://doi.org/10.1007/11802372_68
[32] Hummel, J., Wolff, R., Stein, T., Gerndt, A., & Kuhlen, T. (2012). An Evaluation of Open Source Physics Engines for Use in Virtual Reality Assembly Simulations. In Advances in Visual Computing (pp. 346–357). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-33191-6_34
[33] Wikipedia. List of Game Engines. Retrieved November 11, 2018, from https://en.wikipedia.org/wiki/List_of_game_engines
[34] Epic Games. Unreal Engine. Retrieved November 11, 2018, from https://www.unrealengine.com/en-US/what-is-unreal-engine-4
[35] Unity Technologies. Unity. Retrieved November 11, 2018, from https://unity3d.com/
[36] Unity Technologies. (n.d.). Manual. Retrieved November 11, 2018, from https://docs.unity3d.com/Manual/
[37] Epic Games. (n.d.). Manual. Retrieved November 11, 2018, from https://docs.unrealengine.com/en-us/

Specification and synthesis of the overall architecture

In this section, we elaborate on how the components compose together to one big architecture. We specify the role of each component and their interactions with the others, as well as components that we considered outside of the project scope.



The Platform Solution

The platform domain is the domain of clients and servers, which acts as a host for the other components in the architecture. The server-side programme of the game, which includes parts of the game engine, runs on the ‘game nodes’ and it provides the in-game connection events. These events are used by the agent system of the Artificial Intelligence component. The client also includes parts of the game engine, so it also receives connection events.

The Game Engine Solution

The game engine’s player controller is used in the game client, it receives input from input hardware. The game engine manages all units, terrain and particles and renders them too. Players have in-game units (avatars) as well, so that’s why the player controller has a line going to the Unit entity from the unit’s component. The physics engine can make use of the shapes and physics information provided by the units. The rendering component also uses the shape data from the units.

The Game Unit Solution

The game units, in essence, provide shapes, textures and physics for the game engine, but some non-playable-characters (NPCs) have a controller component. This component is an agent for the artificial intelligence component.

Artificial Intelligence Solution

The AI component uses an agent solution, in which all NPCs have an agent. NPCs are controlled by the game server programme on the server side, They receive events generated by other NPCs, but also events generated by the platform, such as player logins so that new players can immediately become a target. Some other events (e.g. bullet hits) are created by the game engine and passed to the AI through the game unit component.

SAAM analysis

To validate the proposed architecture a SAAM analysis is conducted. This is a scenario-based evaluation method were it is tested how resilient the architecture is against possible changes in the requirements.

Scenarios

As is common in a SAAM analysis [38][39], two kinds of scenarios are considered; direct and indirect scenarios. Indirect scenarios will require the architecture to be modified, for example, “Players want to play the game in virtual reality”. Direct scenarios are already supported by the current architecture for example; “Players want a game unit for a new kind of car”.

Code	Description	Classification	Stakeholder
N1	Players want an AI that they can cooperate with rather than defeat.	Direct	S1
N2	Players want to play the game in virtual reality.	Indirect	S1, S3, S6
N3	Players want to play against each other on a local area network.	Indirect	S1
N4	Players want to be able to create custom units for the game.	Indirect	S1, S3
N5	Players want a new strategy to complete game levels.	Direct	S1, S6, S13
N6	Players want to be able to compete against other players.	Direct	S1
N7	Scenario designers want to offer different physics environments. (e.g. The moon)	Direct	S5, S6, S13
N8	Game developers want to offer weapon components as purchasable content. (e.g. Scope, Silencer)	Direct	S5, S6
N9	Players want vehicles that have mounted weapons.	Direct	S1, S6, S13
N10	Players want to be able to communicate with other players.	Indirect	S1, S5
N11	Players want a highscore to compare the performance of players per stage.	Indirect	S1, S5
N12	Players want to create their own levels.	Indirect	S1, S13

N13	Game developers want to sell custom animations for in-game actions.	Direct	S5, S6
N14	Players want to be able to repair degraded items.	Direct	S1

Scenario interactions

Scenarios that could change the proposed architecture are discussed here.

N2 Supporting virtual reality would have a massive impact on the architecture. First, the selected game engine would need to support virtual reality. Furthermore, the architecture of the rendering engine would need to support the smooth rendering of three-dimensional output, high-resolution and a high framerate. Next, the platform needs to support ultra-low latency and immediate reactions to change in input (such as the position of the head) to prevent nausea. Finally, VR specific input methods would need to be supported.
The game architecture was designed in such a way that it is possible to add the required support for the game engine. However, the latency requirements might require a larger part of the games computation to take place on the client side.

N3 The current architecture does not support playing local games using a LAN connection. If the game developers want to support this, a version of the server architecture should be made deployable for players. They could then create their own server and use it to host local games.
The question is whether the company actually wants to allow this. It would require all the payment options to still go to the server of the game company, while the gameplay should go to the local server.

N4 The architecture is set up to allow the game developers and scenario designers to create game units using an editor. To allow players to create their own custom levels, a (simplified) version of the editor should be made available to the players. This would not require the architecture to be changed.

N10 The game currently does not include a communication channel. To include this in the game, the client-server connection protocol needs to be adapted to include text messages or voice messages. Furthermore, the input should be adapted to support text or voice input. The graphics should also be adapted to include a chat interface that shows who is communicating and their text messages. The architecture does not have to be structurally changed to support these features.

N11 To create a highscore, first a scoring method should be defined. This scoring message should be based on the information that is already stored when a game state is saved. The only difference is that it should also keep track of the historical state of the game, not just the most recent state. Furthermore, a graphical interface needs to be added to the startup menu that shows the highscores per game stage.

This feature does not require a restructuring of the game's architecture. The variability management techniques included in the architecture are enough to support it.

N12 If players want to create their own levels, the changes required are similar to the changes required for players to create their own game units. A version of the level editor should be made available to the players. An additional change is that the information about which game units are available to create a level needs to be available to the player. This can be accomplished by maintaining a connection to the server, or by including this information into the scenario designer that is made available to the players. There should then also be an option to retrieve the newest list of available game units from the server if there has been an update to this list.
This feature would require an adaptation of the scenario editor and the communication protocol, but no structural changes to the architecture are required.

Evaluations

Overall, it was quite hard to think of scenarios that would require the architecture to be changed. The indirect scenarios that we did come up with, can mostly be implemented without fundamental changes to the structure of the architecture, only requiring adaptations of individual parts. This shows that the architecture is already resilient to many possible scenarios that may change the requirements of the project. This is not remarkable, considering that for each solution domain the adaptability requirement was carefully considered, and was implemented such that the model presented would be resilient to changes in the future. This can, for example, be seen in the proposed architecture for the AI component. Rather than a hardcoded model, the design allows for easy modification of goals and actions that the AI can perform. Additionally, the planner that computes the actions to be taken to reach certain goals is implemented as an interface, thus allowing different implementations for the planning algorithm.

Nevertheless, one can never be prepared for everything and no architecture is resilient to all possible changes. For example, if a completely new type of gaming platform is released, it is unlikely that the architecture will be able to be deployed to such a gaming platform. However, it should be possible to adapt the architecture to changes that are somewhat similar to the technologies analysed during the design of this architecture.
For completely new technologies, the analyses made during this design process will be insufficient. These new technologies could offer better alternatives and as such, it is likely that a new product will be more cost-effective than trying to adapt this product to such new technologies.

Lessons learned and conclusions

Lessons Learned

Various lessons were learned during the project that made us more aware of the required skills to create a complete system architecture.

Iterating the project
A big part of creating a solid and complete architecture is setting up valid requirements and concerns, from which the solution domains and eventually the architectures flow. A struggle in the beginning, however, was setting up valid requirements based on the small assignment description. This made us think very broad about the subject and set a wide range of requirements. During the discussion sessions with our supervisor, we were often informed that multiple iterations of our project could solve this problem, and that is one important aspect of software architecture we learned. When we started to focus the scope of our game, requirements started to become obsolete or had to be changed. With every next step we made during the project, we had to go back to previous steps and adjust them to the newly created goals. Mainly the final iteration over the whole project was important after the solution domains and architectures were defined since this iteration also made sure traceability was assured.

Traceability to avoid loose ends
This brings us to our second lesson learned: traceability. It is important to substantiate your choices since the goal of the complete system architecture is to convince people to fund your project in a real-life scenario. Creating traceability made us focus on our line of thought, and gave us a structured way to avoid loose ends.

Communication between group members
Another important factor to create a properly synthesized architecture and to assure traceability is communication between various group members, especially during the solution domain stage. Even though we were able to define separate solution domains, there is still a large interdependency between them. This meant streamlining different solutions to assure the solutions could be connected in the end. Also, certain choices would have become problematic since the game engine of choice, for example, only supports certain AI architectures. Communication between the members made sure these potential problems were solved before they became a problem, by taking each other's choices into account for the individual architectures.

Creating an architecture is not a real-life implementation
The final big lesson we learned, was the fact that creating a system architecture does not mean the same a creating an actual implementation. Most of us have a background in computer science and we started to structure the individual architectures as an actual software implementation. This quickly became a problem since too much detail in classes and functions made the connection of different objects difficult. We learned to step back and focus on the bigger picture: creating an architecture from which software engineers can

create actual code. Here the focus lies on defining important features and connecting various parts of the architecture in a well-defined way. Creating the basic structure of an architecture is here key.

Conclusion

In this report, we described an architecture for a war game. The game had to feature weapons and/or vehicles, roads and buildings. We identified market trends and established other requirements. We decided that we wanted to create a shooter game, as they are popular nowadays. Money needs to be earned after all. From the requirements list, we defined concerns that our architecture should address. The problem domains were directly identified from the concerns and solution domains were identified. The most important solution domains are elaborated in detail in the individual sections and were researched by team members through a literature study. All this traceability and substantiations make us confident that, after all the iterations, the architecture solves the problems from the problem domains.

Extensibility
The game can be updated to include new features in the future. For example, Graceful degradation can be added in a future version of the game because the architecture has been designed to support degradation related features in future revisions. For example, Game Units can have degradationState' value which can be used by the physics component and AI component to implement different behaviours of the unit. Additionally, there is flexibility in the platforms that are supported. We can support new client platforms with little effort as long as they are supported by the engine. The game can also be easily extended with new characters, weapons, levels and worlds. This is because Game Units are not hardcoded, and are instead dynamically instantiated on runtime. Also, other components, such as the AI have flexible goals and actions. Overall, we designed our architecture with flexibility and adaptability in mind. Especially taking care that the future evolutions of the game, as described in the assignment, can be easily supported in revisions of the game.

References

[1] 2018 Video Game Industry Statistics, Trends & Data - The Ultimate List. (2018, November 06). Retrieved November 11, 2018, from https://www.wepc.com/news/video-game-statistics/

[2] Gaming Market Size, Share & Trends Analysis Report By Device (Console, Mobile, Computer), By Type (Online, Offline), By Region (North America, Europe, APAC, LATAM, MEA), And Segment Forecasts, 2018 - 2025. (n.d.). Retrieved November 11, 2018, from https://www.grandviewresearch.com/industry-analysis/gaming-industry

[3] Global Games Market Revenues 2018 | Per Region & Segment. (n.d.). Retrieved November 11, 2018, from https://newzoo.com/insights/articles/global-games-market-reaches-137-9-billion-in-2018-mobile-games-take-half/

[4] Share of digital games revenue by monetization 2017 | Statistic. (n.d.). Retrieved November 11, 2018, from https://www.statista.com/statistics/821451/distribution-digital-games-market-revenue-monetization-model

[38] Slashnode. (n.d.). Retrieved November 11, 2018, from http://slashnode.wikidot.com/seng4420-lect07

[39] Lo, P. (n.d.). *Software Architecture Analysis Method (SAAM) Method (SAAM)*. Lecture. Retrieved November 11, 2018, from http://www.peter-lo.com/Teaching/U08182/L07A.pdf