De Montfort University

Literature Review

Multiplayer Real Time Simulated Card Game

Connor Rowland

P16204361

24/10/2018

CONTENTS

[INTRODUCTION: 2](#_Toc3676296)

[ABOUT DOBBLE: 2](#_Toc3676297)

[SYMBOL GENERATION: 2](#_Toc3676298)

[2D PACKING ALGORITHMS: 3](#_Toc3676299)

[PSEUDO CODE: 4](#_Toc3676300)

[FIRST HEURISTIC: 4](#_Toc3676301)

[SECOND HEURISTIC: 4](#_Toc3676302)

[DIAGRAMS: 5](#_Toc3676303)

[FIRST HEURISTIC: 5](#_Toc3676304)

[SECOND HEURISTIC: 5](#_Toc3676305)

[ENGINE ARCHITECTURE: 5](#_Toc3676306)

[NETWORKING: 9](#_Toc3676307)

[CONCLUSION: 11](#_Toc3676308)

[BIBLIOGRAPHY: 12](#_Toc3676309)

# INTRODUCTION:

The multiplayer real-time simulated card game is based on Dobble. It’s a network-oriented concept, which includes 2D packing algorithms, and finite projective planes to generate cards with a certain number of specific symbols on each of them.

# ABOUT DOBBLE:

Dobble is a card game that was released in 2009, it has a simple rule set and simple mechanics which means all ages from 6 onwards can play and enjoy it. It is best played with 2-8 people. Players compete to see who can find the matching symbol between two cards (asmodee, 2009). There’s fifty-five cards in the deck. Every card in the deck is unique, only sharing a single symbol with every other card. This is because if some cards shared more than a single symbol then the player with that card might have a slight advantage. For example, in the original Dobble, where each card has 8 symbols, if a player has a card with 2 matching symbols, and the other players only have cards with a single matching symbol then the player with 2 matching symbols is more likely to find a matching symbol first, as they have a higher chance to do so. So, there cannot be any bias when creating the symbols on the card (asmodee, 2009).

# SYMBOL GENERATION:

The original Dobble game may have used mathematical methods, based on finite projective (plane) geometry, to generate the symbols on each card, or the symbols could’ve been hand crafted (manually picked) (Bourrigan, 2014).

To generate the symbols, a finite projective plane could be used. The points on the plane would represent the different cards, and the lines going through the points would represent the (different) symbols on each card (ypercube, 2011).

There’s only a few projective planes that have been classified, and the results known, this includes: one, two, three, four, five, six, and eight pictures on a single unique card, where every card has a single symbol that matches. These scenarios work because they make a perfectly square incidence matrix, as they’re prime powers, whereas if the number of symbols on each were seven then it’d be impossible to make it perfectly square, this was proved by Gaston Tarry, a French mathematician (Lam, 1996).

The number of symbols on a card effects the number of cards generated and total symbols needed in the deck. An equation can be used to find out how many symbols are needed and how many cards will be generated, depending on how many symbols are on each card. This equation is , where n is the number of symbols on the card minus one (Bourrigan, 2014).

If a card only had a single symbol on it and kept to the rules of Dobble (no two cards can be the same), then there’d only be a single card in the deck. If there were two symbols on a card, then there’d be three cards in the deck. Three symbols on a card is an interesting scenario, as it would require seven cards, this creates the Fano plane (Bourrigan, 2014). Four symbols on a card generate thirteen cards. Five symbols on a card generate twenty-one cards. Six symbols on a card generate thirty-one cards. And finally, eight symbols on a card generate fifty-seven cards, which is what the card game Dobble would use, but the creators must have removed two cards for some reason, so there’s only fifty-five cards (ypercube, 2011).

# 2D PACKING ALGORITHMS:

To generate a pack of cards that have eight symbols transformed pseudo randomly on each card there needs to be some way to ensure all the symbols fit properly on the card. There’s a single algorithm that stands out. However, this algorithm has many different heuristics that can be used to pack circles into other 2D shapes. The problem that needs to be solved is to pack many unequal circles into a single large circle, of a fixed size, so the following section will focus on that.

The circles on the card will be scattered, without a clear pattern, using quasi-random sequences. This increases the distance variety of “randomness”, as quasi-random sequences have a self-avoiding property, which ensures they don’t cluster, and fill the area efficiently, compared to pseudo random numbers (Cook, 2009). This helps to space the circles out, better, at the start, helping the algorithm place them more efficiently.

The algorithm works by having a center-point that the circles gravitate to, to get as close as possible (Rutten, 2016). This is done by iteratively moving the circles in the direction of the center-point, while checking to see if they overlap with any of the other circles, as soon as an overlap is made/found, the circle stops moving and should be moved apart from the other circle so it’s only touching it (Rutten, 2016) (Aaronasterling, 2010). Once every circle is as close as possible and has stopped progressing closer because another circle is in its way, or it’s reached the center-point then further optimizations can be made, to improve how the circles are packed (Rutten, 2016).

One heuristic uses a generator to find points within the packed circle, closer to the center-point than the furthest away circle, where other circles might be able to fit. It iterates over theses points, checking the distance between the point and the center-point (Aaronasterling, 2010). Either once a single point or every available point has been found the outermost circle[s] can be tested, to see if they fit within the area around that point, if they can then they’re moved there. If the single point method was used, then it will continue to search for other points, until it repeats itself, (looks at the same point twice) as it searches in a linear fashion this should only happen once every point has been checked. However, if every possible point has been found then the points within that area needs to be removed from the list, as it’s already been checked (Rutten, 2016).

Another heuristic that could be used to optimize the global layout of the circles is another type of iterative improvement operator. It arbitrarily chooses a line through the center-point or an area of circles and mirrors the circles at an axis perpendicular to the chosen line, testing whether improvements were made, to the position of the circles (Aaronasterling, 2010).

## PSEUDO CODE:

### FIRST HEURISTIC:

Set center point of the card

Circle Position = SpeudoRandomNumber(Within the card’s area)

While there’s a circle that isn’t in the card’s area

Loop through each circle on the card

If the circle is at the center point or if it keeps colliding with another circle

if the circle is within the card’s area

stop moving it

else

select another spot the circle can be placed

if there’s no spots the circle can fit, and the regenerated circle value is 5

shrink the circle

else

regenerate the circles

else

Move the circle towards the center point

### SECOND HEURISTIC:

Set center point of the card

Circle Position = SpeudoRandomNumber(Within the card’s area)

While there’s a circle that isn’t in the card’s area

Generate an arbitrary line through the center point, of the card

Mirror the circles at an axis perpendicular to the line

If the circles are closer (center – center point)

Keep changes

else

if number of attempts is equal to max attempts

regenerate circles

else

repeat

## DIAGRAMS:

### FIRST HEURISTIC:

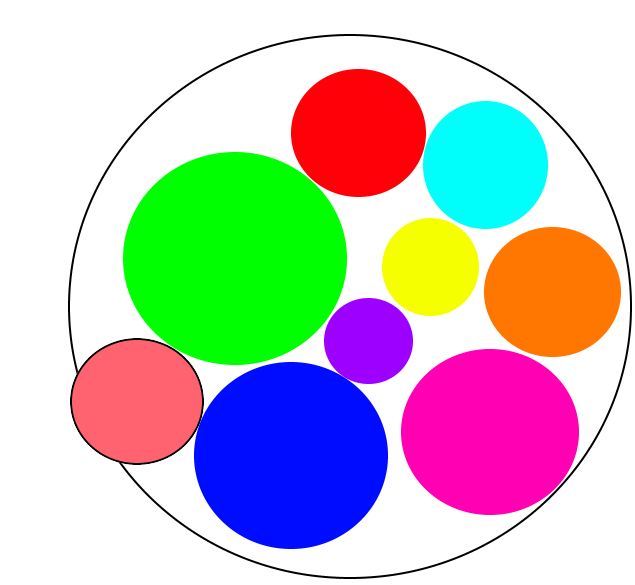
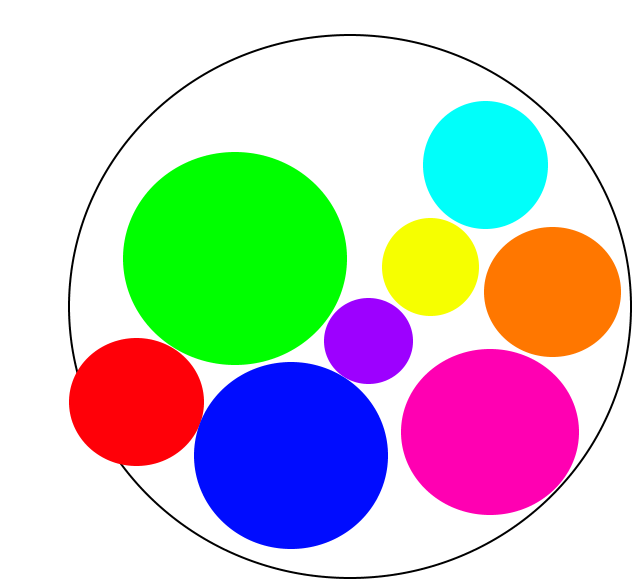


Figure 1 – Circle isn’t within the card’s area. Figure 2 – A position it found, for the circle, within the card.

### SECOND HEURISTIC:

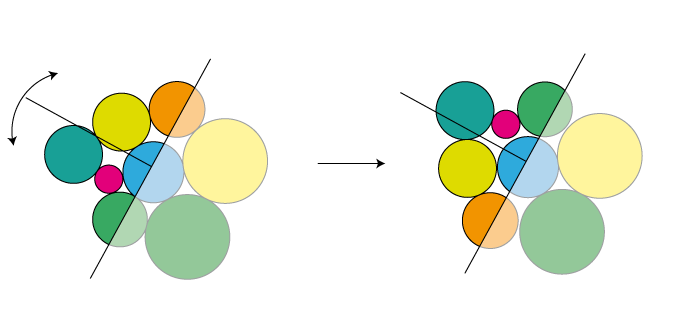


Figure 3 – A line is chosen through the center. Figure 4 – Circle mirrored through at a perpendicular axis.

# ENGINE ARCHITECTURE:

Deciding what engine architecture, a game will be based on is a big technical aspect of the planning. There’s two main structures that have been used in the industry, depending on the project, these types include: an inheritance-based architecture or a component-based architecture. Which of these architectures are used for a game depends on many factors, including: budget, game genre, and development time (Gregory, 2018).

The first of these two architectures to be used widely is the inheritance-based architecture, this is because object-oriented programming (OOP) came before and inheritance is fundamental to it (OOP), and was recognized by many programmers, it was the new easy thing to that allowed for drastic code reuse, and it wasn’t difficult to implement. Before this, programmers would copy and paste code, as function and procedure calls were expensive (Lowe, 2015).

The idea behind an inheritance-based architecture is to capture semantics in a classification hierarchy, like a taxonomy, ordering concepts from generalized to specialized, grouping any concepts that are related in subtrees (Lowe, 2015). This leads to deep hierarchies, where specialized classes inherit from a lot of different generalized classes (West, 2007). This can be seen in figure 5, which displays an animal inheritance tree.

This architecture has many advantages and disadvantages, due to its nature. The advantages include: reusability of code, as classes inherit attributes and behavior from parent classes (Rajput, 2015), but this can be a disadvantage, when the code base gets bigger and more complex, as a class might need to inherit unnecessary attributes it won’t use, to get access to certain ones within that parent class, this leads to either a class inheriting a lot of useless attributes, or the programmer copy and pasting the code the child class needs from the parent class, creating duplicated code; overriding – changing some of the behavior it’s inherited from a parent class, specifically for this child class, this is done through polymorphism; it can make the application code more flexible, as it can use polymorphic data structures (Rajput, 2015); and it’s easier/cheaper to design and build a game based the inheritance approach as, it’s quicker to implement features at the cost of reusability (Gregory, 2018). However, this approach also has many disadvantages, which include: the parent and child classes become tightly coupled, which leads to difficulty in the future, if the parent class is modified, it might break things in the child class, as the child relies on the parent class (Nystrom, 2014) (Rajput, 2015), the deeper the hierarchy the bigger the issue is; due to the nature of inheritance it can have an impact on the performance of the application, as the design of the memory isn’t very cache friendly, so the CPU tends have an increase of cache misses, compared to a component design (Nystrom, 2014); and it suffers from specific problems, such as: member slicing, and the diamond of death problem, which are both difficult to work around in a massive code base.

Member slicing happens when a derived class object is assigned to a base class object. The additional attributes of the derived class object are sliced off to form the base class object. This can be problematic when access is needed to those extra attributes.

The diamond of death problem occurs when a child class uses multiple inheritance and is derived from two class that inherit from the same (base/parent) class. The issue is the child class doesn’t know which attribute[s] to use from the parent of the parent classes (Nystrom, 2014).

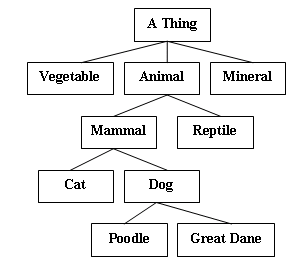


Figure 5 – An inheritance tree for animals (Javanomicon, 2012).

The component-based approach has started to be used over inheritance, it’s gaining popularity, and has been implemented in the latest game engines, as it offers many advantages over inheritance-based architectures (West, 2007). Shallow inheritance can and is normally used in a component-based design, but deep hierarchies are avoided, as inheritance has many great advantages a component-based system can take advantage of.

There’s two principal types in a component-based architecture: Entities, and Components. Entities have a list of components, which provide the entity with attributes and behavior, it’s essentially a container of components. A component is a self-contained system that adds some functionality to an entity (West, 2007), it isolates domains by having each domain be its own component (Nystrom, 2014). A component’s functionality shouldn’t depend on any other component[s] and shouldn’t possess any complex logic (Stein, 2017).

There are different styles of the component architecture, including the entity component system (ECS), which is used in game development (Stein, 2017). This style has many different variations though, but two main ones, where the changes are drastic. The different variations resolve how the components are implemented, communicate, and the structure of the components in the memory.

One variation stores each component type in its own large contiguous block of memory, and then performs the necessary tasks on each component type. This is very cache friendly, as large blocks of components can be read in and then processed, minimizing cache misses (Gregory, 2018).

Another variation groups components into classes, to create different objects, which allows the class to handle the communication between the different components it stores. This increases the coupling and isn’t as cache friends, as the different component types are mixed with each other.

Independent manager objects, known as systems, manage the entities and components life-cycle. Because the entities, components, and systems are (so) independent of each other communication tends to be difficult, but there’s many different ways this can be implemented, a common method is to have an event-manager to distribute events, giving them all a way to communicate (Stein, 2017) (West, 2007).

As this is gaining popularity, and is replacing the traditional inheritance architecture, within game engines, it might seem obvious that there’s many advantages to using this architecture, which include: reusability, once a component has been created it can be reused to create new engines, new systems, new games, and to prototype new concepts (Gregory, 2018); it’s scalable, as new components, entities and systems can easily be added; it’s memory/cache efficient, as the nature of the design allows for big blocks of components to be loaded into the CPU cache and memory, which increases the overall performance of the application (Stein, 2017); and the design is more flexible, as the code is much better decoupled than the inheritance approach.

The component-based design has a few disadvantages, which include: Added complexity, as an object is really a container of objects, which need to be initialized and correctly wired; designing a way for components to communicate can be difficult, especially in large systems; and depending on how the memory of the components is managed there can be a lot of pointer chasing, which may lead to poor performance, as it has a level of indirection.

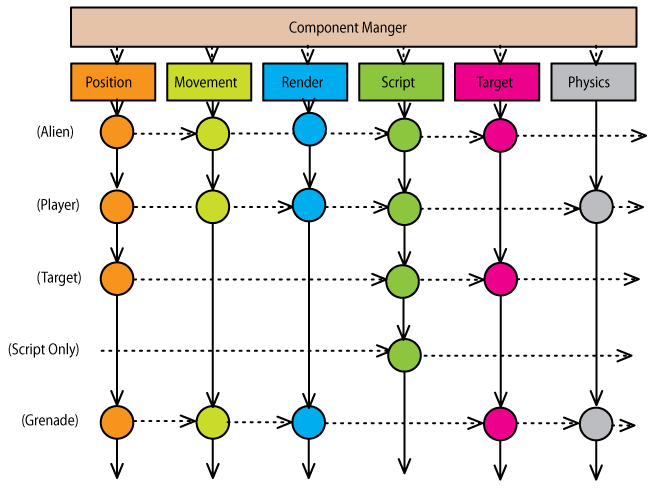


Figure 6 - A simple entity component system design (West, 2007).

# NETWORKING:

In real-time games where time critical data needs to be sent and received with the lowest latency a design decision needs to be made, on what model should be used, whether to use: user datagram packets (UDP), transmission control protocol (TCP), or a mixture of both (Fiedler, 2008).

UDP is a connection-less based model, which means a virtual connection is not automatically established between the parties that are communicating, this is something that must manually be implemented by the programmer (Fiedler, 2008). UDP has a couple of advantages over TCP, it has a lower latency and can be configured similar (for a targeted objective) to TCP, in terms of reliability and flow control. However, TCP still has many advantages over UDP. It is connection based, so it has guaranteed reliable and ordered delivery, it automatically breaks up the data into packets, it handles flow control to ensure it does not congest a network, and it is easy to use. But, there is a latency cost for all these features it has, and it can act unpredictably in certain scenarios. For example, if TCP sends a packet, it gets lost, then resends the packet and that becomes lost. TCP might decide the packet loss indicates network congestion and then it might back off, and wait, which is awful for real-time games (Fiedler, 2008). TCP also uses Nagle’s algorithm (IBM, 2015), which reduces the number of packets that need to be sent, by waiting until there is a certain amount of data in the buffer (waiting to be sent) before sending it (Cheshire, 2005). This makes TCP more efficient, lowering congestion, at the cost of latency. However, it can be disabled on a per-socket basis (Cheshire, 2005).

Using a mixture of UDP and TCP is something that sounds great but when put it into practice it does not work as well as expected. When both protocols are present one of them affects the performance of the other. UDP’s packet loss is especially affected by TCP’s traffic and its flow control mechanism, as TCP flow control continues to increase its window size until packet loss occurs if the window size is large enough(Sawashima , et al., 2016)**.**

Games will mostly use the UDP approach, even though TCP may seem like an obvious choice, as UDP’s only real advantage is a lower latency. However, multiplayer games need to minimize latency as much as they can, so with the reliability overhead of TCP, and its (sometimes) unpredictable behavior, UDP seems favorable (Fiedler, 2008).

Another design decision that needs to be made is what network architecture model should be used to structure the communication between the players. Whether it is a synchronous lockstep with peer-to-peer connection or a client/server model that should be used.

The peer-to-peer lockstep model was used in the early ages of online gaming, and still is in most RTS games, having each node exchange information with every other node, in a fully connected mesh topology (Fiedler, 2010). This network architecture model has many strengths which include: reduced latency due to direct client-client nature of the system, rather than a client-server-client roundtrip for messages, reduced bandwidth because of its deterministic nature, which also makes cheating even more difficult because if a player cheats then they become desynchronized and the other players will be informed, and it doesn’t have a single point of failure, as the game can continue going on if a client or the host disconnects or becomes desynchronized (Terrano & Bettner, 2001). However, there is quite a few weaknesses, such as: with the increase of active nodes there is a higher potential of failure and a higher latency potential (Terrano & Bettner, 2001), and every players’ latency is the same as the player with the highest latency, as the game has to be played out identically on all of the players’ computers, so they have to wait for all of the players’ commands before they can simulate that turn (Fiedler, 2010).

The game also must be deterministic, as mentioned above, this means given the same initial condition and the same set of inputs the simulation gives the same result. This is difficult to simulate because of the differences in floating point behavior between compilers and operating systems (Fiedler, 2014). Once achieved it is much more efficient, especially for large RTS games, which is why it is still used. Instead of having to send a large amount of game data, such as the objects position, velocity, and health, only the player’s input needs to be sent, such as a key input, from the player’s keyboard. There’s still a noticeable latency, while playing the game. The game plays audio and animation feedback instantly to minimize this.

Modern real-time games, excluding most RTS games, use a client/server network architecture. Instead of every player communicating with every other player, they’re now a “client” and communicate with a single computer known as the “server”. This removes the fact that the game must be deterministic because the game only exists on the server. In this version the server has authority over the game, as that is where the game exists. This network architecture has two slightly different variants: a pure client/server variant, and a client-side prediction variant (Fiedler, 2010).

The first variant that came into existence is the pure client/server model, which was used in DOOM and the original Quake. In this model the client (player) doesn’t run any game code locally, but instead sends the player’s key/mouse inputs to the server, which updates the player’s character in the world and responds with a packet containing the state of the player’s character, and any other characters around him/her. An illusion of smooth movement is created by interpolating between the sent updates, creating the networked game (Fiedler, 2010). Latency was still a big problem with this model, even though it only depended on the connection between the client and the server, opposed to it depending on the player with the highest latency, and the client/server set up reduced the bandwidth required on average per-player, as not much data needed to be sent back and forth between them (the client and the server) (Fiedler, 2010). The issue with this is that the client still had to wait for the input to be sent, received by the server, the server to process it, and then for the server to reply, before the game did anything.

Modern FPS games, such as Call of Duty: Modern Warfare, use the client-side prediction variant. In this model the client doesn’t just wait for the response from the server, but instead runs more code than it had before, allowing it to predict movement of the player’s character, in response to the player’s input, locally, and immediately (Fiedler, 2010). When the player inputs something, that input is sent to the server but, now it is also being processed on the client. As mentioned earlier the server is authoritative, because if it was not then the client could easily cheat; for example, he/she could easily lie about his/her position (Fiedler, 2010).

Due to the client[s] predicting their actions, and the server being authoritative an issue arises, when the server’s reply is received by the client the update is in the past, due to latency, if we were to correct the client’s simulation (position/action), with the past data then there would be no point in having client-side prediction, as it is always undone (Fiedler, 2010). To resolve this issue a circular buffer of past character state and input for the local player on the client is maintained. When the client receives a correction from the server it clears the buffer of any state that’s older than the correction and replays the state starting from the corrected state back to the present time, using the inputs stored in the circular buffer (Fiedler, 2010). The only time the client’s state should have to be corrected is when it is affected by something external to the local player’s input (being shot by another player, something the player cannot predict, as they don’t have enough information, when simulating the game locally), or if the player is attempting to cheat (Fiedler, 2010).

# CONCLUSION:

In conclusion, generating which symbols go on what card would be best done once, for each deck size that’s playable (21, 31, and 57 cards), as it will always be the same, but the symbol signs that are placed on the card will be randomly selected/generated, and then randomly placed on the card. This ensures there’s a greater sense of “randomness” and mystery when playing. To ensure all the symbols are on the card and are as tightly packed as possible using a generator to find points circles can fill will give better results and performance, as only certain spots the circle can fit in will be checked, and once an area has been checked it can be removed from the list, having a worst case of: O(n).

Either of the engine architecture types would be suitable, but a component-based approach is probably more beneficial in this case as it allows for great flexibility, easier prototyping, and it will be a simple ECS, as there won’t be many components. It will also offer a performance boost, as cache misses will be less frequent.

A client server-based approach, using a custom implementation, built on top of UDP would work best, giving the best performance and ensuing players’ communications are handled as soon as possible, with the lowest latency. However, with the time frame I have, I believe a TCP implementation might be more suitable, with Nagle’s algorithm disabled, so data is sent immediately. Considering the data being sent, the amount of data, and pace of the game I don’t believe a custom UDP implementation would be necessary, either.

# BIBLIOGRAPHY:

Aaronasterling, 2010. *Position N circles of different radii inside a larger circle without overlapping.* [Online]   
Available at: https://stackoverflow.com/questions/3851668/position-n-circles-of-different-radii-inside-a-larger-circle-without-overlapping  
[Accessed 09 11 2018].

asmodee, 2009. *Dobble.* [Online]   
Available at: https://www.asmodee.co.uk/featured-product/dobble/  
[Accessed 13 11 2018].

Bourrigan, M., 2014. *DOBBLE AND THE FINISHED GEOMETRY.* [Online]   
Available at: http://images.math.cnrs.fr/Dobble-et-la-geometrie-finie.html?lang=fr  
[Accessed 12 11 2018].

Cheshire, S., 2005. *TCP Performance problems caused by interaction between Nagle’s Algorithm and Delayed ACK.* [Online]   
Available at: http://www.stuartcheshire.org/papers/NagleDelayedAck/  
[Accessed 03 11 2018].

Cook, J. D., 2009. *Quasi-random sequences in art and integration.* [Online]   
Available at: https://www.johndcook.com/blog/2009/03/16/quasi-random-sequences-in-art-and-integration/  
[Accessed 09 11 2018].

Fiedler, G., 2008. *UDP vs. TCP.* [Online]   
Available at: https://gafferongames.com/post/udp\_vs\_tcp/  
[Accessed 28 10 2018].

Fiedler, G., 2010. *What Every Programmer Needs To Know About Game Networking.* [Online]   
Available at: https://gafferongames.com/post/what\_every\_programmer\_needs\_to\_know\_about\_game\_networking/  
[Accessed 29 10 2018].

Fiedler, G., 2014. *Deterministic Lockstep.* [Online]   
Available at: https://gafferongames.com/post/deterministic\_lockstep/  
[Accessed 29 10 2018].

Gregory, J., 2018. *Game Engine Architecture.* THIRD EDITION ed. s.l.:A K Peters/CRC Press.

IBM, 2015. *The tcp\_nodelay or tcp\_nagle\_limit options.* [Online]   
Available at: https://www.ibm.com/support/knowledgecenter/en/ssw\_aix\_71/com.ibm.aix.performance/tcp\_nodelay\_tcp\_nagle\_limit.htm  
[Accessed 03 11 2018].

Javanomicon, 2012. *Javanomicon01 - Inheritance.* [Online]   
Available at: http://monkeys.imaginary-realities.com/index.php?title=Javanomicon01\_-\_Inheritance#Inheritance  
[Accessed 24 11 2018].

Lam, C. W. H., 1996. *The Search for a Finite Projective Plane of Order 10\*\*\*.* [Online]   
Available at: http://www.cecm.sfu.ca/organics/papers/lam/  
[Accessed 12 11 2018].

Lowe, S., 2015. *Composition vs. Inheritance: How to Choose?.* [Online]   
Available at: https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose  
[Accessed 18 11 2018].

Nystrom, R., 2014. *Game Programming Patterns.* FIRST ed. s.l.:Genever Benning.

Rajput, S., 2015. *What are advantages of inheritance in C++?.* [Online]   
Available at: https://www.quora.com/What-are-advantages-of-inheritance-in-C++  
[Accessed 18 11 2018].

Rutten, D., 2016. *Sample: Circle Packing Plug-in (VB.NET).* [Online]   
Available at: https://wiki.mcneel.com/developer/sdksamples/2dcirclepacking  
[Accessed 07 11 2018].

Sawashima , H., Hori , Y. & Sunahara , H., 2016. *Characteristics of UDP Packet Loss: Effect of TCP Traffic.* [Online]   
Available at: https://web.archive.org/web/20160103125117/https://www.isoc.org/inet97/proceedings/F3/F3\_1.HTM  
[Accessed 29 10 2018].

Stein, T., 2017. *The Entity-Component-System - An awesome game-design pattern in C++.* [Online]   
Available at: https://www.gamasutra.com/blogs/TobiasStein/20171122/310172/The\_EntityComponentSystem\_\_An\_awesome\_gamedesign\_pattern\_in\_C\_Part\_1.php  
[Accessed 20 11 2018].

Terrano, M. & Bettner, P., 2001. *1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond.* [Online]   
Available at: http://www.gamasutra.com/view/feature/131503/1500\_archers\_on\_a\_288\_network\_.php?page=2  
[Accessed 29 10 2018].

West, M., 2007. *Evolve Your Hierarchy.* [Online]   
Available at: http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/  
[Accessed 18 11 2018].

ypercube, 2011. *What are the mathematical/computational principles behind this game?.* [Online]   
Available at: https://stackoverflow.com/questions/6240113/what-are-the-mathematical-computational-principles-behind-this-game  
[Accessed 12 11 2018].