POLISH-JAPANESE ACADEMY
OF INFORMATION TECHNOLOGY

Faculty of Information Technology

Department of Multimedia

Multimedia – Game Development

Mariusz Hausenplas
ID number 10638
Arkadiusz Kalbarczyk
ID number 10450

Development of an advanced real-time multimedia application in Unity environment based on the game project "Laboratory Night"

B.Eng. thesis
Written under supervision of
Daniel Sadowski, MSc.

Warsaw, January 2016

POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Multimediów

Multimedia – Programowanie Gier

Mariusz Hausenplas
Nr albumu 10638
Arkadiusz Kalbarczyk
Nr albumu 10450

Tworzenie zaawansowanej multimedialnej aplikacji czasu rzeczywistego w środowisku Unity na przykładzie projektu gry komputerowej "Laboratory Night"

Praca inżynierska
Napisana pod kierunkiem
Mgr. inż. Daniela Sadowskiego

Warszawa, styczeń 2016

Streszczenie

Niniejsza praca prezentuje proces tworzenia zaawansowanej multimedialnej aplikacji czasu rzeczywistego w środowisku Unity na podstawie projektu gry „Laboratory Night”. Opisane zostało teoretyczne ujęcie zagadnień związanych z przygotowywaniem graficznych gier komputerowych, jak i sposób implementacji w omawianym projekcie przy użyciu dedykowanej technologii Unity. Praca została podzielona na cztery części. Pierwsza część przedstawia wstęp oraz nakreśla cele i założenia autorów. Druga część zarysowuje teoretyczne aspekty wykorzystanych technologii. Trzecia część opisuje proces planowania oraz implementacji projektu. Czwarta część zawiera podsumowanie i wnioski po zakończeniu przygotowywania projektu.

Table of contents

1. Introduction section 5
   1.1. Structure 5
   1.2. Goals 5
2. Theoretical section 6
   2.1. History of computer games 6
      2.1.1. Introduction 6
      2.1.2. Earliest computer games 6
      2.1.3. Age of commercialization 7
      2.1.4. Modern computer games 9
   2.2. The Unity game engine 10
      2.2.1. Introduction 10
      2.2.2. History of Unity 11
      2.2.3. Unity as a game engine 13
   2.3. Mesh technology 14
      2.3.1. Introduction 14
      2.3.2. Polygonal mesh 14
      2.3.3. Low Poly modeling 15
      2.3.4. Skinning 16
      2.3.5. Mesh technology tools 17
   2.4. Texture mapping 18
      2.4.1. Introduction 18
      2.4.2. Basics of texture mapping 18
      2.4.3. Additional texture effects 18
      2.4.4. Corrective methods and texture filtering 21
   2.5. Lighting 24
      2.5.1. Introduction 24
      2.5.2. Methods of Illumination 24
      2.5.3. Perception of Light 27
      2.5.4. Lighting environment 27
   2.6. Shadow mapping 28
      2.6.1. Introduction 28
      2.6.2. Light mapping 28
      2.6.3. Shadow volumes 28
      2.6.4. Shadow maps 29

2.7. Rendering 29
2.7.1. Introduction 29
2.7.2. Rendering techniques 30
2.7.3. Real time rendering 32
2.8. Shaders 36
2.8.1. Introduction 36
2.8.2. Shader types 36
2.8.3. Shader Model 5.0 37
2.9. Particle systems 38
2.9.1. Introduction 38
2.9.2. Design 38
2.10. Collision detection 39
2.10.1. Introduction 39
2.10.2. Bounding boxes 39
2.10.3. Bounding spheres 39
2.10.4. Mesh-to-mesh collisions 40
2.10.5. Collider combinations 40
2.11. Artificial intelligence 40
2.11.1. Introduction 40
2.11.2. Navigation and pathfinding 41
2.11.3. Finite state machines 41
3. Design and implementation section 43
3.1. Design 43
3.1.1. Game design 43
3.1.2. Team and choice of technology 44
3.1.3. Project and source code organization 46
3.2. Implementation 47
3.2.1. Controls 47
3.2.2. Basic gameplay logic 48
3.2.3. Meshes 49
3.2.4. Texturing 49
3.2.5. Lighting 50
3.2.6. Shadow mapping 53
3.2.7. Rendering 53
3.2.8. Shaders 54
3.2.9. Particle systems 54

3.2.10. Collision detection 55
3.2.11. Artificial Intelligence 57
3.2.12. Sounds 58
4. Conclusion section 60
4.1. Final overview 60
4.2. Appendix A – CD Content 61
4.3. Appendix B – Game User Manual 62
5. Bibliography 63
6. Index of figures 65

1. Introduction section

The following thesis aims to describe the process of developing an advanced real-time multimedia application based on the Unity engine. Trying to provide a full, comprehensive view on the project's creation, this document covers both theoretical aspects of the spectrum of technologies, as well as describes the usage in the actual application.

1.1. Structure

This document was divided into four sections, starting with the Introduction Section being a broad overview of the entire project, as well as describing the work's goals and motivations. The Theoretical Section focuses on examining the technologies commonly used in the development of applications similar to the one being the subject of this thesis. The Design and Implementation section expands on topics presented in the previous part, providing information on how these technologies were applied in the project, as well was describing the usage of engine-specific tools which proved necessary during the development process. Finally, the Conclusion Section contains a general summary of the work and describes some final conclusions arisen after finishing the application's creation.

1.2. Goals

The goal of this thesis is to provide a comprehensive view on the development of a real-time multimedia application in the form of a video game called “Laboratory Night”. As the project was created in an advanced graphical environment offered by the Unity engine, this work naturally aims to describe crucial components of such modern applications from both the theoretical and practical perspectives.

Although much of the game's development revolved around the choice and utilization of the Unity engine, this thesis also aims to give an overview of other aspects vital to the successful creation of a similar application. These include story design, team assembly, scheduling or setting up certain source code quality standards – all of these constituting an important part of pre-development stage.

2. Theoretical section

2.1. History of computer games

2.1.1. Introduction

The concept of a computer game is generally understood as a computer program, which, by defined means of human-computer interaction, generates appropriate feedback, usually displayed on some video device. Earliest examples of interactive games involving electronics technology include the "Cathode ray tube[1] Amusement Device" (1947), in which a player, by means of a knob, controls the movement of a dot displayed on a screen. After having positioned it in an appropriate place within a time limit, he or she may, by pressing a button, simulate an explosion in which a strong CRT impulse brightens the screen

The history of computer games is strongly related to the history and the development of computer hardware. The emergence of early digital programmable machines such as ENIAC[2] (1945) and EDSAC[3] (1949) enabled the scientists to perform various complex computations utilizing the benefits of their ability to be reprogrammed in order to solve arbitrary computational problems. Initially being restricted to government-run military laboratories, these early computers became accessible to the wider public as a result of their introduction to various scientific centers and, in particular, academic institutions conducting research on early computation theory, artificial intelligence and human-computer interaction.

2.1.2. Earliest computer games

It is difficult to precisely track down the development of first *de facto* games written for early programmable machines, as many of such project may have been used for simple demonstrative purposes in the fields of military or general science, thus lacking appropriate documentation or any surviving evidence of their existence. A change occurred upon the creation of the game OXO (1952), written for the EDSAC computer as a part of Alexander S. Douglas'[4] Ph.D. thesis on human-computer interaction. OXO was a Noughts and Crosses simulator containing the earliest known example of a graphical interface displayed on a monitor, showing player's and computer's moves on a 35x16 pixel

[1] Cathode ray tube (CRT) – an image display technology utilizing vacuum tubes with an electron gun used to create images on fluorescent screens.
[2] ENIAC (Electronic Numerical Integrator And Computer) – the first digital programmable built in 1946 by the Moore School of Electrical Engineering at the University of Pennsylvania.
[3] EDSAC (Electronic Delay Storage Automatic Calculator) – early digital programmable computer built in 1949 at the University of Cambridge Mathematical Laboratory.
[4] Alexander Shafto Douglas (1921-2010) – a British professor of computer science affiliated with University of Cambridge.

CRT screen. The desired move was entered using a telephone dial, to which a replying move was selected by an artificially intelligent machine. As a result of its strictly academic character, OXO did not gain particular attention outside the Cambridge University. On the contrary, the game Tennis for Two (1958), although never released commercially, was well received on the day of its première on October 16, 1958 in Brookheaven National Laboratory[5]. Tennis for Two required two players to compete in a table tennis simulation game, using specially designed controllers. Remarkably, the ball's movement path was rendered taking into account its apparent velocity and drag, resulting in a relatively realistic and convincing simulation. Despite not having attracted publicity outside the Laboratory, Tennis for Two proved computer games may gain considerable attention in the future and develop into a new and unique branch of software.

Further improvement of computer technology has led to the establishment of academic institutions focused on research of the new area of science. Specifically, the Artificial Intelligence Laboratory of Massachusetts Institute of Technology proved to have been crucial for the rise of early 'hacker culture', involving student groups being granted a possibility to write programs to be run on machines operating in the university. Aiming to discover the boundaries of a newly-introduced PDP-1[6] machine, Steve Russell[7], a member of one such group, has written a game Spacewar (1962), in which two players shoot each other's starships, at the same time being pulled to the star located in the center of the monitor. The program rapidly gained popularity among universities throughout United States, inspiring students and staff to write extensions and even entirely re-model the game. Furthermore, the manufacturer of PDP-1 decided to include Spacewar in their machines sold with the Type 30 Precision CRT Display, treating Russell's program as a diagnostic tool, which contributed to the game's recognition.

2.1.3. Age of commercialization

By the end of the 1960s the prices and sizes of computers began to decrease, allowing for further experiments on the development of more widely-available games. The spread of Spacewar and Sega-manufactured[8], commercially successful and visually rich electromechanical arcade game Periscope (1966) has led Nolan Bushnell[9] to the conception of Computer Space (1971), the first

[5] Brookheaven National Laboratory – a laboratory located in Upton, New York conducting research on nuclear physics, material physics, chemistry and environmental biology.
[6] PDP-1 (Programmed Data Processor–1) – digital programmable computer constructed in 1959 by Digital Equipment Corporation.
[7] Steve Russell (born 1937) – an American computer scientist affiliated with Massachusetts Institute of Technology.
[8] Sega Corporation – video games manufacturer based in Japan, founded in 1940.
[9] Nolan Kay Bushnell (born 1943) – an American engineer and entrepreneur, a key figure in the early video game industry.

widely-distributed coin-operated arcade computer game, deployed on a dedicated machine. Despite the gameplay being strongly based on that of Spacewars, Computer Space turned out to be a commercial failure due to its high complexity and poor marketing.

Continuing decline of hardware prices and rise of television-driven entertainment industry has inspired various companies to pursue the goal of combining computing units and television displays, thus aiming to construct the earliest home video game consoles. Ralph H. Baer's[10] Odyssey (1972), manufactured by Magnavox[11]. Originally shipped with 12 games basing on shooting, quiz and table tennis mechanics, the console reached the sales of 100 000 within the first year of its release, making it a commercial failure. Less than a half a year after the release of Odyssey, Bushnell's newly founded Atari Incorporated[12] released the highly acclaimed table-tennis simulator Pong (1972), based on one of the games included in the Magnavox' console. Pong's unprecedented success inspired other companies to create similar arcade games, including Gun Fight (1975), being one of the first to simulate gun combat.

The increasing popularity of computer games resulted in a greater variety of products. Colossal Cave Adventure (1976) is an example of an early text-based adventure game, while the advancement in video displays gave rise to games involving colored graphics. The release of immensely popular Space Invaders (*see Illustration 1*) in 1978 marks the beginning of the golden age of arcade video games, characterized by the spread of arcade machines, rapid development of game-manufacturing companies and gain of mainstream media attention.

Furthermore, early 1980s saw the release of classic titles such as Pac Man (1980), Frogger (1981) and Donkey Kong (1981) which achieved global distribution, making the industry reach a revenue of $11.8 billion [Chesebro89] in its peak in 1982.

Illustration 1. Screenshot from the game Space Invaders.

[10] Ralph Henry Baer (1922-2014) – an American engineer and video game developer, a key figure in the early video games industry.
[11] Magnavox – electronics manufacturer based in the USA, founded in 1917, acquired by Philips in 1974.
[12] Atari, Inc. - video games manufacturer based in the USA, founded in 1972, disbanded in 1984.

With the release of numerous, often low quality games saturating the market, and the rise of powerful home computers, the American game industry experienced a major crash in 1983, with the total revenue falling to $5 billion [Loguidice14]. As companies started to recover by 1985, the computer game development shifted towards the modern machines such as Commodore 64, ZX Spectrum or Atari ST, as well as the newly arrived 3rd generation, 8-bit consoles, often identified solely with the Nintendo[13] Entertainment System (NES). Introduced in 1985, the console was sold bundled with the game Super Mario Bros, becoming an instant success. Other significant games include The Legend of Zelda (1986), Dragon Quest (1986) and Final Fantasy (1987), all of them evolving into internationally-recognized role-playing-game franchises.

2.1.4. Modern computer games

Marked by the release of 4th generation, 16-bit consoles and modern personal computers, the 1990s saw the unprecedented revolution of computer graphics, paving the way for the development of fully three-dimensional games, dramatically increasing the games' realism. This innovative technology resulted in the conception of new genres, such as the First Person Shooter and the advent of 3D fighting games or 3D racing games. Early examples of popular products include, respectively: Wolfenstein 3D (1992), Tekken (1994) and Ridge Racer (1993), featured on both PCs and consoles such as Sega Genesis (1989) and PlayStation (1994). Moreover, while the traditional arcade games having significantly declined from the consoles or PCs, modern hand-held devices took over the arcade market. Released for Nintendo's Game Boy, games such as Tetris (1989) or Super Mario Land (1989) assured the popularity of casual games.

Continuing development of hardware resulted in further increase of graphics and sound quality, greater color depth and resolution. The growing availability of Internet has given rise to a number of on-line games, including the successful Massively Multiplayer Online Role-playing Games (MMORPG), most famously represented by World of Warcraft (2004), having almost 10 million active players [BlizzardPress].

Furthermore, due to the popularity of mobile phones, the mobile games market has emerged and solidified upon the release of smartphones, powerful enough to render advanced 3D graphics. Other important trends aiming to improve the user experience include the introduction of motion-capturing devices in which the player's body moves are tracked using an external motion controller (Wii Remote for the Nintendo Wii console, Sixaxis for the Sony[14] PlayStation 3 console) or a camera

[13] Nintendo – video games and gaming consoles manufacturer based in Japan, founded in 1889.
[14] Sony Corporation – electronics and video game manufacturer based in Japan, founded in 1946.

(Sony PlayStation Eye, Microsoft[15] Kinect), as well as the autostereoscopic[16] graphics found in the Nintendo 3DS console.

2.2. The Unity game engine

2.2.1. Introduction

Throughout a major part of video games history, the development of graphically advanced multimedia applications has relied on hardware-close programming using low-level languages. Technologies such as assembly[17] and later C or C++ offered speed and flexibility required to perform complex operations generating graphics. Due to the nature of work involved, a high level of expertise in these technologies was usually required. At the same time, none of the modern, high-level programming languages such as Java or Python seemed to have brought enough adaptability and efficiency to be suited for computation-heavy tasks involving rendering graphics. As a result, most development teams ended up with custom solutions called game engines spanning thousands of lines of source code only serving as a backbone to implement an actual working application. The increasing popularity of computer games and rising numbers of would-be game creators led companies to create proprietary engines that would contain many ready-made components and simplify the overall development process. Nowadays, Unity stands as one of the most successful and commonly used example of these.

In simple terms, Unity is a game engine and a cross-platform game integrated development environment[18] used to develop video games for all major platforms including the PC, PlayStation 4, iOS or Android. Founded in Copenhagen, Denmark, the product evolved into one of the most popular game engines of the industry, holding 45% of the market share *(see Illustration 2)*. The idea behind Unity was to create an affordable game engine with a flat learning curve, aimed primarily at non-experienced developers and small, independent studios, yet quickly gained major attention and was utilized by companies from the entire spectrum of the video games industry as well as numbers of amateur programmers.

[15] Microsoft Corporation – operating systems, utility software and hardware manufacturer based in the USA, founded in 1975.
[16] Autostereoscopy – a display technology enabling to simulate the illusion of three-dimensional depth without the use of specialized glasses.
[17] Assembly language – low level programming language interacting directly with the memory. Its implementations vary based on machine's architecture.
[18] Game integrated development environment (game IDE) - special environment solely made for game development, usually bound to a single game engine.

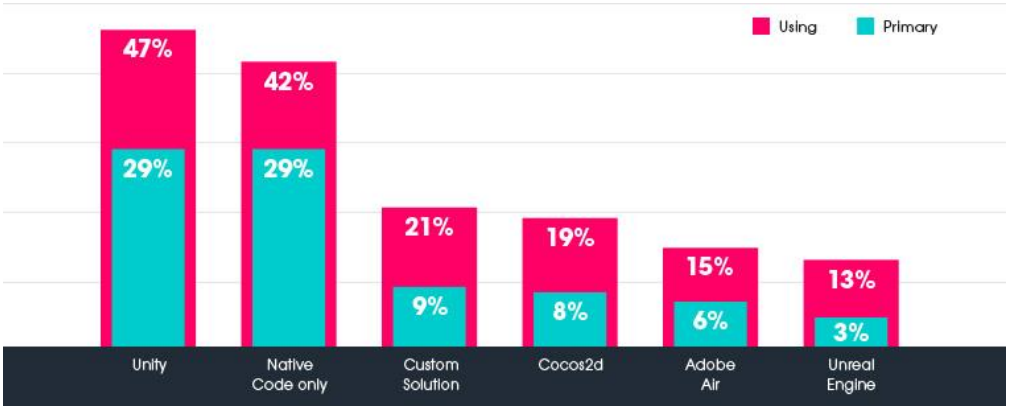

Illustration 2. Market share of game engines.

2.2.2. History of Unity

History of the engine can be tracked down to year 2002, when a Danish programmer, Nicholas Francis decided to ask for an assistance via Mac OpenGL message board, needing help with a shader[19] system intended to be applied in his custom game engine. A reply came from Joachim Ante, who at the time lived in Germany. Both programmers agreed on developing a common system that could fit their independently-developed engines, shortly after deciding to combine their engines into one. Shortly after launching the project, Francis and Ante acquired a third companion. Upon learning about the concepts behind what to be Unity, David Helgason decided to participate in the venture, and soon became the CEO. Initially the company incorporated under the name Over the Edge Entertainment (OTEE).

With no prior experience managing a company OTEE's board decided to mimic a business model designed by a British gaming company Criterion[20]. According to their practices, product should first earn recognition from reputable game developers, before targeting casual customers. Criterion, however was not Unity's only inspiration. At the early stage of development, both the engine's user interface and user experience was highly influenced by the much-praised Apple Final Cut Pro[21].

After having worked on Unity for almost two years the engine had to undergo a thorough testing phase. Consequently, the Danish programmers decided to develop a first fully commercial game built upon their engine. As a result, it not only provided valuable feedback for Francis and his colleagues but also aided them financially. The game called Gooball was published in 2005, through

[19] Shader – piece of source code containing instructions for graphics card how to provide certain level of color and special image effects.
[20] Criterion – British game developer from Guildford, Surrey mostly known for Burnout series.
[21] Apple Final Cut Pro – originally developed by Macromedia Inc. video editing software, provided with dynamic editing interface, magnetic timeline, multichannel audio editing and many other features.

Ambrosia Software[22]. With the achieved profits OTEE was able to hire developers to fix the engine before its initial release. Furthermore the company reincorporated into Unity Technologies. Unity's first version 1.0.0 was release during Apple's Worldwide Developers Conference.

At the time, Unity was still unrecognized by vast majority of gaming industry, it mostly drew attention of independent developers and hobbyists. Initially rooted in the Mac OS X community, Unity 1.1 offered support for both Microsoft Windows and web browsers in order to target much broader audience. Idea of supporting web browser games was a turning point for the company, as until then there was no serious alternative for Adobe's Flash[23]-based games.

Significant income of Unity Technologies allowed the enterprise to recruit new employees for the project, thus leading to milestone release of Unity 2.0 in 2008. It not only offered extended Windows support but also directly utilized the Microsoft DirectX[24] API[25]. Following newly arising market trends, a version of Unity dedicated for iOS system was released. Finally, the year 2008 marked a breakthrough due to a decision made by Cartoon Network to use the engine for the development of MMORPG[26] game named FusionFall, bringing major attention to the establishing technology.

Year 2010 brought a third major update to version 3.0. Its most significant features involved editor unification and low level debugging possibilities. Majority of existing development editors from various platforms were combined into a single product *(see Illustration 3).* Further improvements of the engine lead to another important release of Unity 3.5 that embedded support for Flash deployment. Recent versions of the engine encompassed even greater concepts. Most important among them were Mecanim[27] animations, 2D support, Shuriken particle system, real time global illumination, audio mixer, PhysX[28] 3.3 and a brand new animation system.

[22] Ambrosia software – American software company focusing on utilities and games
[23] Adobe Flash – software platform accessible on Windows, OS X and Linux. Uses ActionScript for games, animations and graphics.
[24] Microsoft DirectX – low level programming interface supporting access to one's computer multimedia capabilities of display and audio cards
[25] API (Application Programming Interface) – a defined set of tools exposed or distributed by one system to be used by others
[26] MMORPG (Massively Multiplayer Online Role-Playing Game) – mixes the genres of role-playing games and massively multiplayer games in which players can interact with each other within a virtual world.
[27] Mecanim – Unity's animation technology that includes tools for creating state machines, automatic retargeting of animations and blend trees
[28] PhysX – a Nvidia technology used to simulate real-world physics behavior of in-game objects.

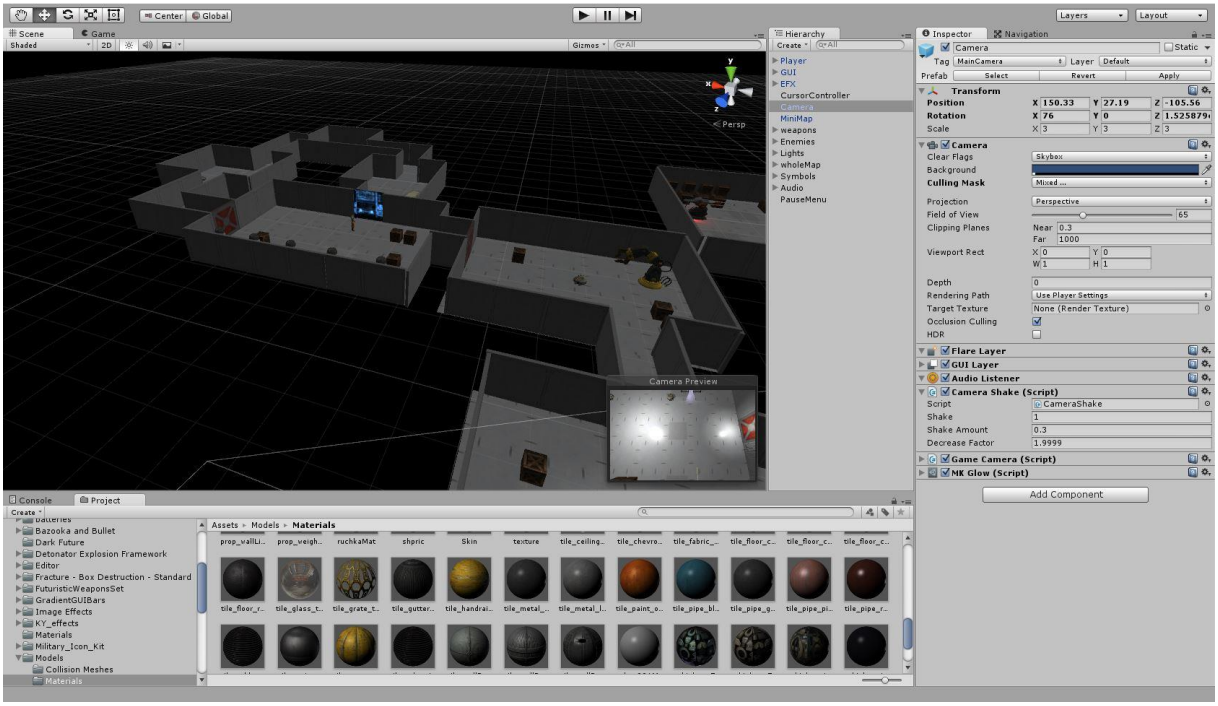

Illustration 3. Unity Editor.

As of 2015 [Somla] there are over 3.3 million registered users along with an estimate of 600 million gamers across the globe. Unity Technologies supports almost all platforms and incorporates of 18 offices with over 470 employees scattered worldwide.

2.2.3. Unity as a game engine

Being a comprehensive game development utility, the Unity engine supports the majority of technologies used to render advanced graphics which are a foundation of modern video games. Among all, the solutions include texture mapping with bumps and normal maps, dynamic lighting and shadows as well as post-processing effects, thus virtually all of the work related to the image generation is delegated to the engine core and is no longer the responsibility of a programmer.

Apart from real-time graphics rendering, Unity provides means to easily shape the gameplay with a unified, platform-independent input-processing system. Intended as a fully multi-platform engine, it is therefore easy to detect and handle user input with a single function call regardless of used controller. This leads to significantly smaller, simpler and more testable software, as source code is not repeated and controlling devices are declared and can be switched on-hand in the built-in Input Manager.

Collision detection is another important gameplay aspect that is taken care of by the engine. Intersections are being calculated based on world-space positions of readily-available components: the Box, Sphere, Capsule and Mesh colliders used as approximations of in-game objects. The

responsibility of the programmer is to attach these components appropriately and handle on-collision events. More importantly, the Unity collision detection system works in close cooperation with the Rigidbody physics simulation. If attached with a Rigidbody component, an object will by default behave according to the rules of the physics system upon a hit, unless it was specified otherwise. This aspect is extremely important as the programmer is yet another time relieved from the need to integrate the two systems, which is often desired in dynamic productions requiring realistic physics.

Finally, one of the most crucial elements of modern computer games is the presence of advanced artificial intelligence and, in particular, the solution to the pathfinding problem. Once again, the Unity engine brings a solution in the form of the Navigation system which enables the developer to generate walkable areas based on existing surfaces as well as to select characters to be movable agents equipped with A*-driven login. The engine goes as far as to support more complex meshes as navigable space (e.g. stairs) as well as define shortcuts in the form of OffMeshLink components, allowing for movement beyond the designated area.

2.3. Mesh technology

2.3.1. Introduction

There exist various methods to represent 3D objects using mathematics. In recent years meshes became increasingly popular and dominated the field of computer graphics. Polygonal meshes are their most popular implementation used in game development.

2.3.2. Polygonal mesh

Polygonal mesh represents a collection of vertices that define a shape of a 3D object. Two vertices describe an edge and at least three edges represent a polygon *(see Illustration 4)*. Due to current 3D graphic cards hardware optimization, polygonal meshes are mostly shaped in a form of triangles.

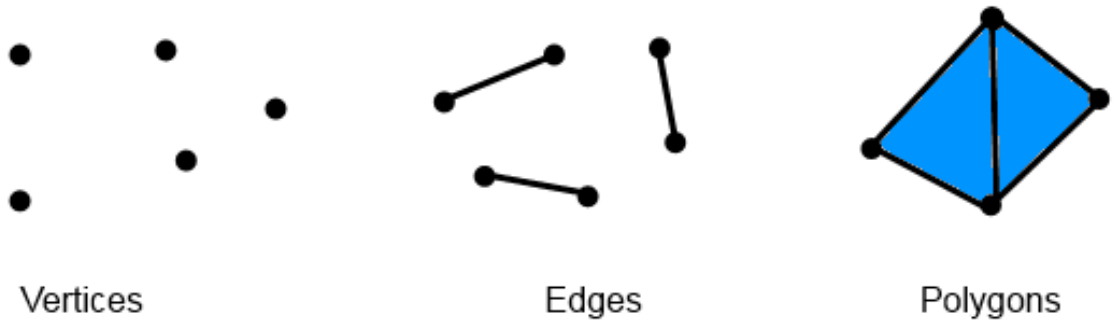

Illustration 4. Structure of meshes.

Triangles describe the surface on which they lie, furthermore there always exists a plane that can intersect all the vertices of a triangle. Additionally, vertexes are capable of storing additional information such as color and texture coordinates.

2.3.2.1. Representations of polygonal meshes

Polygonal meshes may be defined in numerous ways, using different techniques to store vertices, edges and polygons. These include:

- Face vertex meshes – collection of vertices with corresponding sets of polygons pointing to them
- Winged Edge meshes – due to explicitly linked structure of the network it allows for quick traversal between vertices, edges and polygons
- Half Edge meshes – similar to Winged Edge meshes, however stores only half the edges traversal information
- Quad Edge meshes – does not store any reference to polygons as they are implicit in representation
- Corner tables – vertices are stored in predefined tables. This representation allows for quick and efficient polygons retrieve. Operation of applying changes to polygons is however slow
- Vertex-vertex meshes – only vertices pointing to another vertices are represented. Due to the simple structure this representation does not allow for efficient operations to be performed on meshes [Tobler06]

2.3.3. Low Poly modeling

With nowadays technology even millions of polygons per scene are not a challenge to modern Graphical Processing Units. However other factors also have to be taken into the consideration, namely game engine that limits number of handled polygons. Gameplay of a modern game must be smooth and interactive, thus meaning that GPU[29] should be capable of rendering scenes at least at a 30 FPS[30] rate. Due to this fact, each scene should limit the number of polygons to the minimum, while trying to maintain the desired level of detail. In order to achieve balance between the two factors, both the artists and programmers have to cooperate closely, as usually game development requires so-called low poly modeling. What it means is that artists should keep meshes simple and restrain themselves from adding too much detail to each scene *(see Illustration 5)*. Hardly ever exact polygon

[29] GPU (Graphics Processing Unit) – electronic circuit altering and manipulating memory to improve generation of advanced graphics.
[30] FPS (Frames Per Second) – a measure of frequency at which consecutive frames are displayed.

numbers are given, but it is most commonly accepted that a typical character low-poly model should consist of around 4500 triangles [CodeConventions].

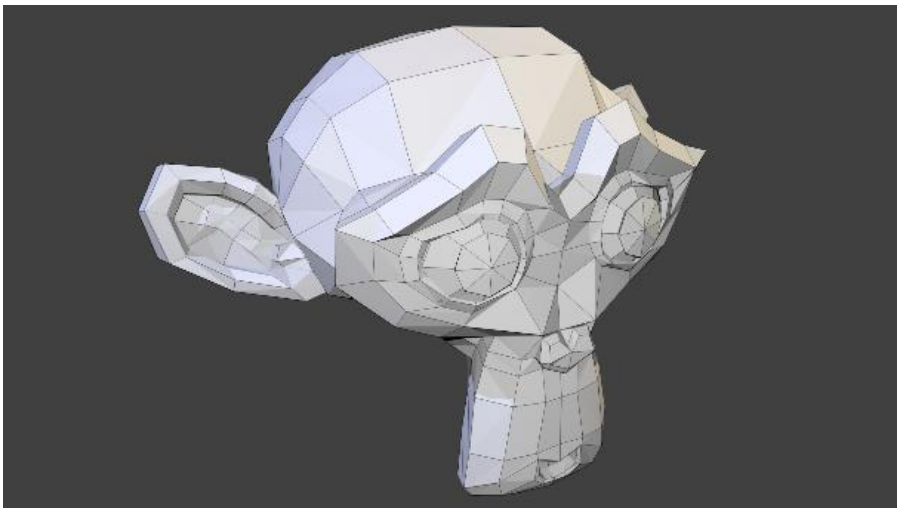

Illustration 5. Model with small number of polygons.

2.3.4. Skinning

The ability to apply skeletons to models is one of the crucial features of 3D tools like Maya, Blender or 3DS Max. This allows for simple animations of objects as opposite to explicit manipulation of every vertex by hand. This technique is referred to as skinning. Three-dimensional skeletons are analogous to human body. Defining joints and bones, they serve the purpose of deforming meshes they are attached to. In order to improve realism, each smaller bone has a parent bone, hence virtual skeletons may be grouped into a tree-like hierarchy *(see Illustration 6).*

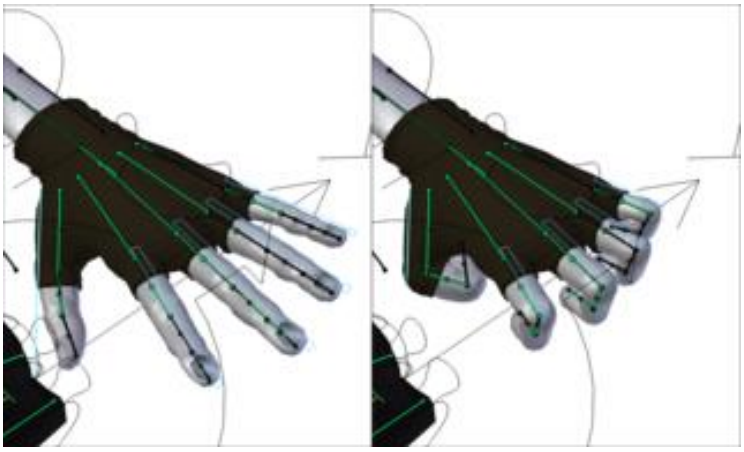

Illustration 6. Simplified hand model from open source project Sintel.

2.3.4.1. Bones and bipeds

A hierarchical set of interconnected bones is used to animate mesh. This approach is often used to animate humanoids, as it seems natural and intuitive. However the same technique may serve a purpose of deforming objects of any shape, including animals or other non-realistic creatures.

Bipeds are the new approach to model skinning. There is a slight difference in philosophy between the two techniques: bipeds use ragdoll[31] systems instead of bones. Although basic principles remain the same, the second approach provides additional features allowing to more visibly overlap parts of the mesh. Furthermore, the whole process is more automatic and simpler than with bones.

2.3.5. Mesh technology tools

Artists nowadays benefit from various animation and modeling software tools, helping them create and animate 3D objects. The two most popular products are Maya[32] and 3DS Max[33]. The reason behind their success is that both of them provide most complete support on all stages of production. On the other hand, they are very expensive, thus mostly used by major companies of movies, games and television industries.

Blender[34] is an alternative product *(see Illustration 7)*, encapsulating most of features provided by Maya or 3DS Max. Due to its open source nature, it is mostly used by independent developers. However, even famous Hollywood productions such as Spider Man 2 used Blender during the preproduction stage.

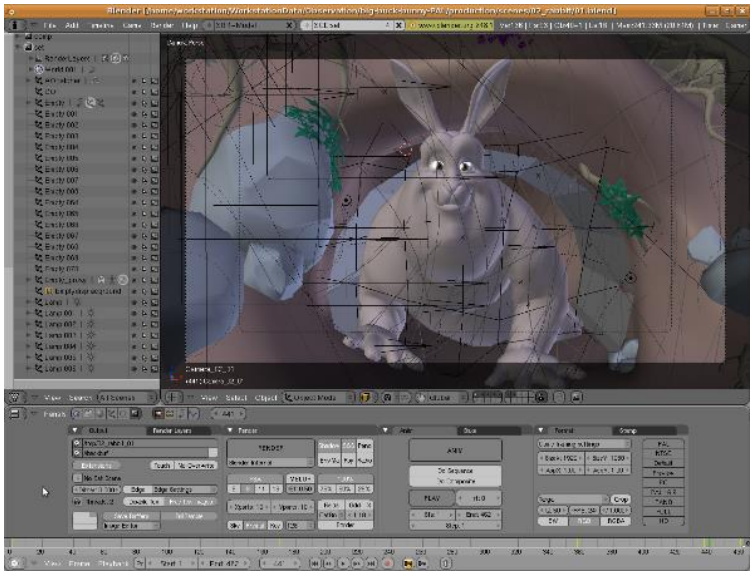

Illustration 7. Blender Editor.

[31] Ragdoll – group of rigid bodies interconnected by a system of constraints limiting each other's movement.
[32] Autodesk Maya – graphics software developed by Alias Systems Corporation
[33] 3D Studio Max – graphics software developed by AutoDesk, Inc.
[34] Blender – open source graphics software created by Blender Foundation

2.4. Texture mapping

2.4.1. Introduction

In the field of computer games development as well as other areas involving real-time graphics rendering, proper shaping and shading of meshes must usually be supported by various other techniques in order to deliver a convincing image. Specifically, for the sake of achieving realistic colors and overall decent outlook of a 3D object it is possible to create a 2D image to be put directly on the mesh. Various techniques used to perform this task are referred to as texture mapping.

2.4.2. Basics of texture mapping

The first step of applying a flat image surface on a complex 3D mesh involves selection of an appropriate object and a texture. When the resources have been collected, the following simple procedure called uv-mapping could be performed. First, it is necessary do define the coordinate system so that at each of the polygon's vertices the 3D object position is described by (x, y, z) axes in screen space and the image is conventionally described by a pair of (u, v) coordinates. Then, the texture is sampled onto the image according to the used method, usually utilizing the Bresenham's[35] Line Algorithm[36]. In other worlds, there needs to exist a correspondence determining the mapping of every pixel of the mesh with a pixel from the planar image. Consequently, one should end up with a pair of (u, v) coordinates for every pixel which could then be displayed with appropriate color value.

Albeit simple and fast, the limitation of this process can be seen when an object is rotated. This is because the plain linear interpolation algorithm does not take into account the object's depth. Certain corrective actions can be made in order to map the sources in perspective to the viewer's position. These are usually referred to as parametric texture mapping. As a result, fragments of a mesh which are closer to the viewer will be enlarged, while the ones farther away will seem contracted, according to the rules of human eye perception.

2.4.3. Additional texture effects

Oftentimes it is desired to make the texture reflect additional visual effects which would simulate somewhat deformed or other 'special' surfaces. A naive approach involves simply re-shaping the mesh according to our needs, but such process would surely take much time and resources. As a result, several methods were developed in order to simulate certain phenomena by introducing special bitmap images. Among others, these include bump mapping, normal mapping and parallax mapping.

[35] Jack Elton Bresenham (born 1937) – American computer scientist working in the field of computer graphics.
[36] Bresenham's Line Algorithm – an algorithm invented by J.E. Bresenham determining what points should be selected in order to approximate a straight line between two given coordinates.

2.4.3.1. Bump Mapping

A typical scenario involves creation of a bumped texture mapping in which the resulting object would seem somewhat rugged and uneven. First proposed in 1978 by James Blinn[37], the Bump Mapping technique enables modifying the 3D model's appearance without actually introducing additional complexity to the mesh itself. Instead, a complementary image needs to be created which would serve as a template for the object's bumpy shape, i.e. define displacements and local changes in the surface's height. This file, called a heightmap, is then used to figure out the object's lighting according to the used reflection model. Typically, for each pixel of the mesh, the surface normal is changed according to the corresponding pixel on the heightmap, as virtually all popular lighting models define the object's reflection by taking into account the values of a polygon's normal vector (*see Illustration 8*).

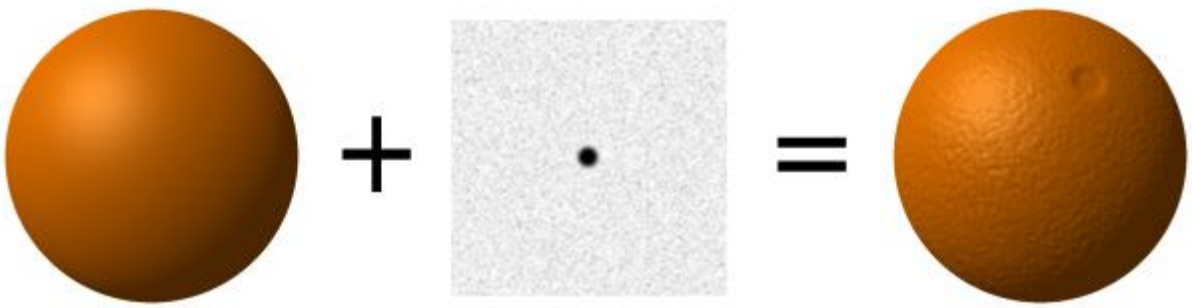

Illustration 8. Bump mapping.

2.4.3.2. Normal Mapping

As an extension of the Bump Mapping technique, another texture-improvement method was developed. This time, contrary to simulating random displacements, the Normal Mapping procedure aims to deliver a much more realistic image by allowing to specify exact 'depth' values to the regular texture *(see Illustration 9).*

[37] James Blinn (born 1949) – American computer scientist working in the field of computer graphics.

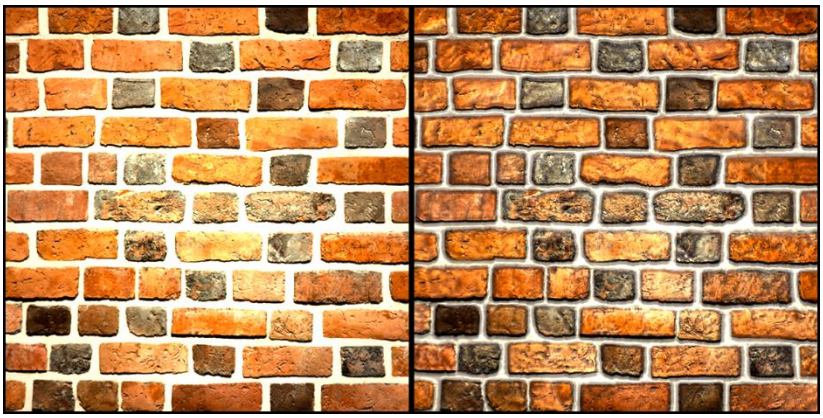

Illustration 9. Object without (left) and with (right) normal map applied.

Instead of calculating the lighting based on each of the polygons' normals, Normal Mapping assumes that an additional file called a *normal map* is created and serves as means for the object's shading. The file contains information on each texel's[38] (x,y,z) coordinates defining its normal vector. It is during the rendering phase when each of the object's pixel's color value is determined combining data from the plain texture and its normal map.

The Normal Mapping technique proves especially effective when applied to low-polygon models, as improvements in the model's realism are notable without greatly stressing the GPU or increasing memory consumption by introducing a more detailed mesh. What is more, a common practice involves generating the normal map derived from the much more complex high-polygon model itself, which is then simplified and has the normal map applied. Thanks to its simplicity and effectiveness, this method has been heavily used in many of contemporary computer games, especially these offering highly detailed, convincing visuals.

2.4.3.3. Parallax mapping

Dedicated heightmaps may also be used to increase overall depth and dynamics of colors in textures that not necessarily need to display the object in a photorealistic[39] way. Parallax mapping technique enables to increase image quality on surfaces viewed at an angle. Typically, the procedure involves matching the texel with a corresponding point on a heightmap, which is then displaced according to a factor defined in the algorithm as well as the current view angle. At steep corners, this displacement value is high enough so that certain points in the texture appear to have greater depth, yielding more convincing images *(see Illustration 10).*

[38] Texel – a single, indivisable point on a texture determining its color value.
[39] Photorealism – a quality of a generated image in which the scene closely resembles real world.

Illustration 10. Object before (left) and after (right) performing parallax mapping.

2.4.4. Corrective methods and texture filtering

Although a texture-mapping procedure on a 3D model is completed with no defects and exactly to the creator's expectations, there is often the case that when viewed at a certain perspective an image may appear somehow deformed and unrealistic. Problems may appear especially when the used texture resolution is inappropriately adjusted, or when an object is looked at a sharp angle. Well-known means of overcoming these issues are mipmapping and various types of linear filtering.

2.4.4.1. Mipmapping

A high-resolution texture mapped to an object viewed from a long distance, or one which is small enough, will be subject to aliasing, i.e. appear to be unnaturally sharp, with obvious distortions and artifacts. One of the most common solutions to this problem is to create a mipmap, that is a series of textures containing the same image, each one being progressively scaled down by the consecutive power of 2 (*see Illustration 11*). Consequently, it is during the rendering phase when an appropriately sized image will be chosen to be mapped to an object. The selection will depend on the 3D model's actual size on the screen. As a result, the smallest possible image needs to be rendered, thus minimizing the number of visual defects.

In terms of efficiency, although an improvement in quality can easily be observed, the usage technique still results in much higher memory consumption, as additional space needs to be allocated for lower-resolution files, even though the rendering itself usually takes less time, as less texels need to be mapped.

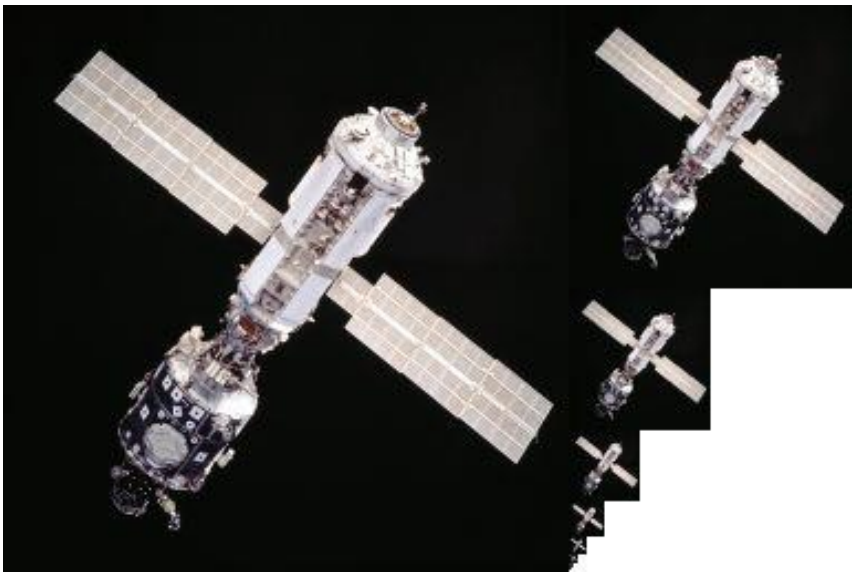

Illustration 11. Sample mipmap.

2.4.4.2. Linear filtering

Obvious image distortions may also appear when a textured object is viewed from a very close distance, or enlarged. After having zoomed in to the right distance, the model will appear to be covered with square-shaped spots representing texels, in the case of a plain raw texture. It is therefore necessary to introduce certain filtering techniques that would improve the image's quality.

A fast, yet rather simplistic approach involves the use of nearest-neighbor interpolation algorithm. This technique compares pixel with the actual texture's texels: each pixel's color is changed exactly to the one of the closest texel in the process called interpolation. As a result, this technique yields quick rendering times, although generated images are still not pleasing, appearing unrealistically sharp when enlarged.

Further texture corrections deliver much better results by aiming to slightly distort the image, rather than simply substitute nearest pixels based on their position relative to the nearest texel. Bilinear filtering involves combining the color values of four-texel groups. For each of the groups, an average is computed, which serves as a foundation for determining the pixel colors, based on the distance from the texels. Consequently, a smooth, slightly blurry transition between colors is generated without rendering visibly contrasting parts as is the case of nearest-neighbor interpolation.

Trilinear filtering, on the other hand, tackles the issue of aliasing which is partly solved with the use of mipmaps. Although mipmapping is generally effective for models viewed from long distances, it may produce some faulty results when both closer and farther objects are being rendered simultaneously and combined with bilinear filtering. Trilinear filtering aims to connect advantages of both methods. First, the regular bilinear interpolation is performed for two textures: ones with low-

and high-resolution. Secondly, another linear interpolation is run to determine pixel values between the previous results. The resulting image is thus relatively smooth, with no aliasing issues in the case of nearby mipmapped textures with different resolutions.

2.4.4.3. Anisotropic filtering

Both bilinear and trilinear filtering techniques are designed to improve the quality of visuals by creating a delicate blur in transitions between areas with different colors. In order to combat this issue and increase the image sharpness when an object is viewed at a sharp angle, the anisotropic filtering method was developed.

This technique considerably improves on the mipmapping procedure by creating even more additional textures with various scaling ratios. Furthermore, the basic image is also reproduced in different proportions. As a result, one may end up with evenly-sized bitmaps as well as others with ratio being, for instance, 2:1 or 4:1, usually denoted as “2x” or “4x” respectively *(see Illustration 12).* The latter ones are used when texture mapping is conducted for objects displayed at steep angles.

Although considered a standard in virtually all graphics-driven computer games, it needs to be noted that anisotropic filtering tends to be intensive for GPU and memory, especially when the maximum scaling factor is as high as 16:1.

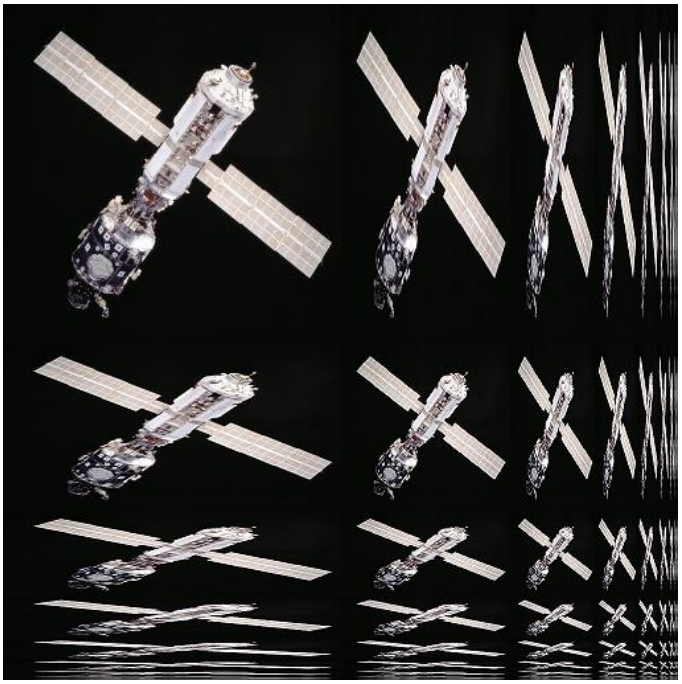

Illustration 12. Sample bitmap used in anisotropic filtering.

2.5. Lighting

2.5.1. Introduction

Right atmosphere and mood are two crucial factors for almost any type of modern video games. Lightning impacts the way players perceive artificially generated game worlds in most popular genres, from FPS[40] to RTS[41]. Lighting merits a study of its own, and requires years of training to master subtleties of color and light effects. Major AAA[42] game studios dedicate whole teams just to the arcane of light effects. The reason behind it is that analysis of light behavior is complicated both in real life and the world of computer graphics. In real world, light travels from the source of emission, and can be reflected, scattered or refracted on every surface it encounters (*see Illustration 13).* For the sake of providing a similar experience, developer attempt to mimic those effects in their games.

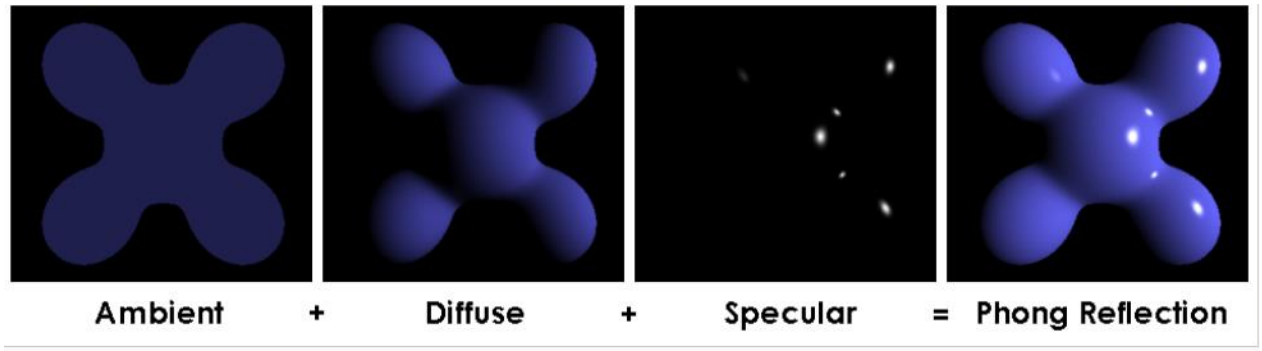

Illustration 13. Different lighting effects.

In order to avoid costly calculations most games use method which sends object as a whole unit to the GPU, where it is mapped and rendered into 3D space. This way, the need of sending millions of rays into a scene is avoided. On the other hand, it is troublesome to mimic e. g. effect of reflection, because in this substitutional approach models do not store information about surrounding objects. Nevertheless this technique exists for a long time, even the graphic cards provide dedicated pipelines aiding the process of drawing objects using this approach.

2.5.2. Methods of Illumination

There exist few standard methods of calculating how light illuminates a surface. Each of the methods has a great number of various implementations. For the purpose of this study, the most common configurations were selected: the Lambert and Phong lighting models.

[40] FPS (First Person Shooter) – a game genre centered on a gun and played from first person perspective
[41] RTS (Real Time Strategy) – a subgenre of strategy games, where player has to control his domain dynamically
[42] AAA – a term classifying games with highest budget for both development and promotion

2.5.2.1. Lambert Light

Lambert Light is a simplistic model dating back to the 18th century, named after its creator Johann Lambert[43]. Lambertian objects emit light evenly across all viewing angles. In other words, objects look the same, despite different viewing positions and angle (*see Illustration 14*).

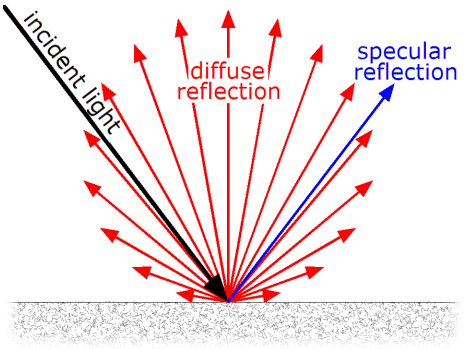

Illustration 14. Diffuse reflection.

Due to its basic structure, Lambert's technique is applied to objects that only have the diffusive light[44], without any reflection or glossiness. In order to calculate lighting of a certain point on an object, particular data about the light and object itself is required. Firstly a relative position of a point to the source of light must be obtained. It is given by a formula:

$$RLP = light_{position} - point_{position}$$

Secondly, in order to get the direction which the surface is facing, a normal vector of the given object is needed. Further calculations are straightforward. Surfaces facing the light directly are fully bright. Brightness intensity of an object decreases linearly as the vector's normal turns perpendicular to the light source *(see Illustration 15)*.

[43] Johann Heinrich Lambert – 18th century Swiss scientist, an important contributor to subjects of mathematics and physics.
[44] Diffusive light – reflection where an incident ray is identically reflected at many angles.

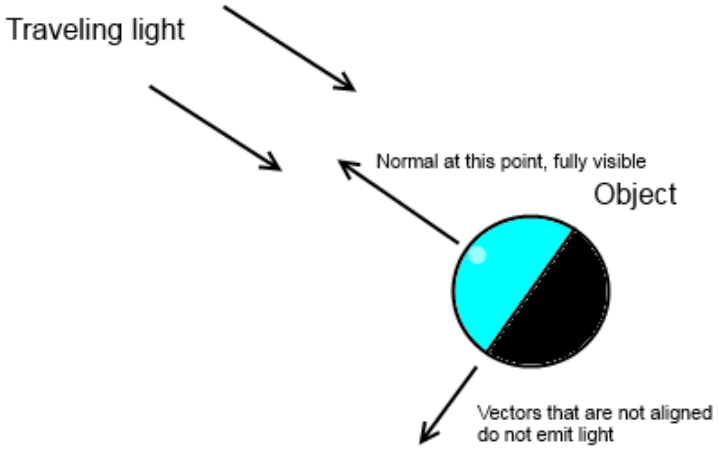

Illustration 15. Lambertian Light.

In order to determine whether two vectors, namely light direction and surface normal face the same direction, an angle between them needs to be computed. It can be obtained using the dot product formula described in the Rendering Chapter.

2.5.2.2. Phong lighting model

A lighting model named after its creator, Bui Tuong Phong[45]. It is usually applied onto specular surfaces, which reflect the light directly off the object. The intensity of reflection can either be bright as on the body of a car, or dull as on a plastic surface (*see Illustration 16*).

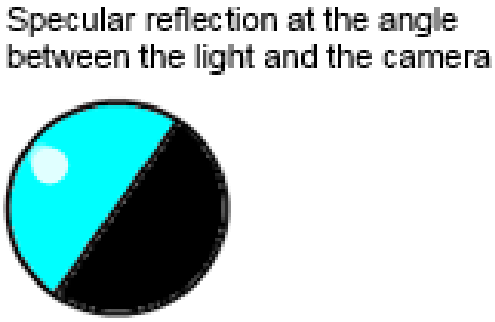

Illustration 16. Specular highlight.

Phong model consists of two parts: diffuse light, the one obtained from Lambertian calculations, and also from the light's reflection. In order to obtain the second component, an angle from which spectator observes an object needs to be known. It then has to be compared to an angle at which the light hits the object. Due to the fact that light reflects at the angle exact to the hitting angle from a shiny surface, it can be easily obtained. Information about the angle of the light from

[45] Bui Tuong Phong (1942 – 1975) – a Vietnamese scientist working in the field of computer graphics

the camera to the object is also required.

The Phong equation takes into the consideration two additional attributes: shininess and specular coefficient. The former is used to describe how big should the highlight be, the greater the value the smaller the highlight, while the latter specifies the rate at which the light reflects, thus providing brightness intensity, as in the formula:

$$specular_{coefficient} * (reflection_{dotView})^{shininess}$$

2.5.3. Perception of Light

As mentioned in the introduction, the way people perceive real world is strongly affected by light. It allows to distinguish shapes and details of objects. It is hardly possible to mimic all of the edges of refraction in a highly detailed surface. Every sharp edge results in a distinct change in object's appearance, thus adding more computations to the rendering process.

2.5.4. Lighting environment

Even seemingly minor changes to the light affect game's environment. In order to create realistic surroundings, a deep understanding of how light interacts with objects is required. Certain genres like survival horror games rely heavily on the light manipulations, one of the most famous examples being the game SOMA (2015) (*see Illustration 17*). Other examples of intricate light-shade play include enemies lurking between the shadows of the darkness and trying to ambush the player or even hidden traps placed around a map.

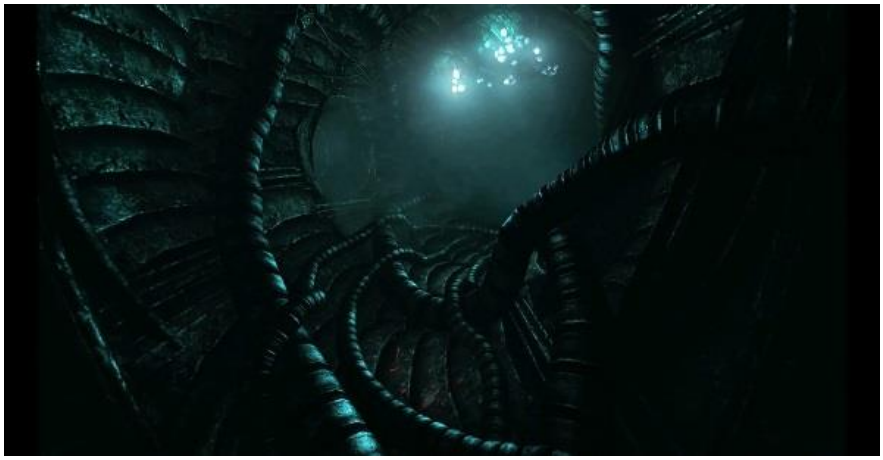

Illustration 17. Screenshot from the game SOMA.

Quality of lighting in video games impacts the gameplay greatly. There exists however no single formula providing ideal balance between colors, shadows and light as it seems to be a highly subjective matter, dependent on the designer's concept. There exist certain techniques to improve the

experience. For instance, storyboards are not only great for planning how the action will progress, but also can be useful for creating proper lighting. During the creative process of planning a game's mood emerges, which then affects the way light is implemented. It is often derived from or even influencing the story or game mechanics, transforming it into an integral part of overall gameplay.

2.6. Shadow mapping

2.6.1. Introduction

As a natural consequence of introducing more complex lighting to the scene, it is necessary to produce shadows as well. In order to maintain an appropriate level of realism, it is nowadays absolutely necessary to properly visualize unlit, faded surfaces when clearly defined sources of light exist. The advancement in development of graphics hardware made it possible for techniques such as light mapping, shadow volumes and shadow mapping to solve the task of generating quality shadows.

2.6.2. Light mapping

Driving inspiration from texturing methods such as normal mapping, the light mapping procedure involves creating a dedicated image called a light map. Introduced in the game Quake[46], this technique allows for effective definition of static shaded surfaces. The image – often in the gray-scale form – would represent how an object would be lit. Specifically, each of the light map's pixels is associated with the texture's pixel. In the simplest variant, these pairs are then multiplied, yielding resulting pixel color values. Usually, the computation takes place prior to the rendering phase, thus generally no real-time lighting modifications are possible.

2.6.3. Shadow volumes

Although light mapping has remained a standard technology for illuminating or shading models in computer games, the development of GPUs allowed developers to utilize certain techniques that would visualize shaded surfaces in real-time. Being one of the first [Crow77] successful dynamic shadow-rendering methods, the shadow volumes method delivers highly convincing results.

The mechanism assumes that the volume is created by first rendering the scene shadowed, then fully lit. Rays are then cast from the light source and those crossing the mesh serve as definition of masks dividing the area into two excluding segments: ones which are shadowed and normally lit. The surfaces need to be "closed" by adding a front- and back-caps, ensuring the shadow volume has clearly defined boundaries. Next, these parts determined either as being affected by lights or shadowed are put in a special data structure called a stencil buffer. The scene is then rendered by

[46] Quake – a first-person shooter game developed by Id Software, released in 1996.

blending the obtained masks.

2.6.4. Shadow maps

On the other hand, the shadow mapping technique, currently a *de facto* standard, tackles the problem of surface shading from a different perspective.

Instead of performing the ray tracing procedure to determine boundaries between faded and illuminated areas, the scene is first rendered from the perspective of a light source and only a dedicated depth map is created. Similar to other helper structures such as heightmaps, the shadow map holds information on all visible surfaces' depth values, i.e. how far away they are from the light source. Then, the scene is rendered from the actual desired camera's position and the *depth test* is performed to determine the proper shading of area in view. The surface fails the test when the "real", camera-sourced depth value is greater than the associated one stored in the shadow map, meaning an object should be rendered shaded. Finally, the whole scene is rendered in line with the results obtained after the depth test.

The use of shadow maps allows for extremely effective shadow projection, yet at a high memory and computation time cost due to the need to store a highly detailed depth map. Should the scene be probed at lower accuracy, the obtained shadows will appear to have obvious distortions, usually in the form of strong aliasing or overall hard edges (*hard shadows*). As a result, various filtering and other workaround algorithms have been developed and are commonly used in contemporary computer games. Some of these include:

- Percentage-close filtering [Reeves87] involving initial creation of several depth maps. Obtained values are then averaged and a shadow is rendered based on the results
- Cascaded shadow maps [NvidiaShadowMaps], especially effective for shading large terrain portions. Several depth maps are created, each of the differing in resolution, yielding less accurate samplings for far-away objects.
- Smoothies [Chan15] which serve as shape extensions used for smoothening of possibly complex shadow-casting models. The final depth map is constructed by blending shadow maps obtained from sampling both the actual object and its complementary Smoothie.

2.7. Rendering

2.7.1. Introduction

It is hard to meet the competition in nowadays game industry. Games have to be appealing to players in both terms of gameplay and graphics. Developers attempt to overcome the competition providing premium quality products and unique features. However majority of their sophisticated

approaches result in performance demanding games. In order to decrease requirements, simultaneously maintaining market standards certain techniques may be implemented.

One of the most important optimization techniques is rendering. By technical means it stands for an engineering program encapsulating selective range of disciples related to mathematics, software development and light perception. It may also be referred to as a process of transforming 2D or 3D models into images. Rendering is one of the most significant topics in computer graphics. It may be applied in a huge variety of areas like video games, movies, visual effects or architecture.

2.7.2. Rendering techniques

It would be barely possible to elaborate on every existing rendering algorithm, therefore several of the more efficient techniques are presented below:

- Rasterization
- Ray casting
- Ray tracing

First and the most popular approach indicates a process, where an image in a vector graphic format is converted into a raster image[47]. Polygons that form 3D scene are rendered onto a computer screen producing 2D surface. As mentioned before, polygons are collection of triangles, where each of them is described by 3 vertices. Therefore rasterization algorithm describes transformation of a stream of vertices into corresponding 2D points and triangles on the screen. Principally this transformation is achieved applying matrix multiplication operation[48]. This process may occur in both directions (*see Illustration 18*). Additionally it is worth mentioning that Rasterization is one of the fastest algorithms, thus making it ideal for real time application, where objects have to respond immediately to the user input and actions.

[47] Raster image - a dot matrix data structure that represents rectangular grid of pixels
[48] Matrix multiplication – a procedure which takes a pair of matrices, and by multiplication of these produces another matrix

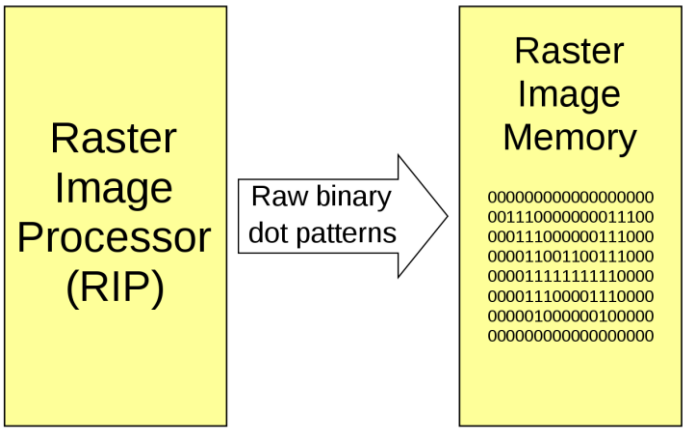

Illustration 18. Process of Rasterization.

In the early age of computer graphics both terms “Ray Casting” and “Ray Tracing” where used interchangeably. More recently, however those terms were distinguished from each other. First of the two mentioned techniques requires only a single calculation to be performed for every vertical line on the screen, making it an efficient approach for rather slower computers. In Ray Casting a game map is a 2D square grid, with each point having either null value or a positive value. Null value represents an empty space, whereas positive value means an object and serves as a reference to its corresponding texture and color. This effect is achieved by sending a ray in the direction depending on players looking direction, corresponding to his x – screen coordinate. The ray travels along with the player until it hits a square with a positive value. Size of the object to be drawn is estimated based on the distance between the player and the given object. The further the distance the smaller the object. Last of the mentioned algorithms is the Ray Tracing method. It is a technique used for rendering graphics along with sophisticated light interactions. In other words it is possible to obtain complex solutions like shadows or transparent surfaces. Despite the stunning features it provides, the implementation is in fact straightforward. The algorithm attempts to simulate the path taken by emitted light photons as they travel across the environment *(see Illustration 19*).

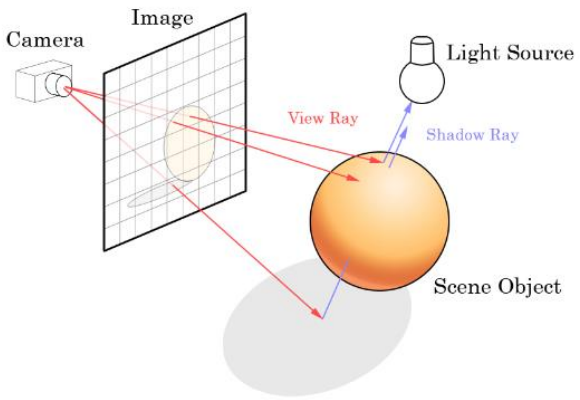

Illustration 19. Path taken by light.

Every ray is tested against every object on the scene to check whether at any point they intersect. Unlike in real world, in ray tracing light is traced reversely, from the viewer's camera to its source of emission. If an intersected object is covered with shadow, a second 'shadow' ray is emitted towards the source of light [Akenine04].

2.7.3. Real time rendering

Real time rendering is a sophisticated technique allowing the user to interact with the virtual environment. It requires both the CPU[49] and GPU to perform its complex calculations. In order to apply real time rendering one must understand few concepts like coordinate system, vectors or matrices.

2.7.3.1. Coordinates system

Within a coordinate system we are able to assign certain amount of values to any point in the space. Number of values correspond with the number of dimensions of the system. A point with zeros assigned to all its coordinates is called a middle point of the system. In order to represent 3D space two coordinate systems can be used:

- Left-hand coordinate system
- Right-hand coordinate system (*see Illustration 20*)

[49] CPU (Central Processing Unit) - electronic circuit performing instructions of a computer program

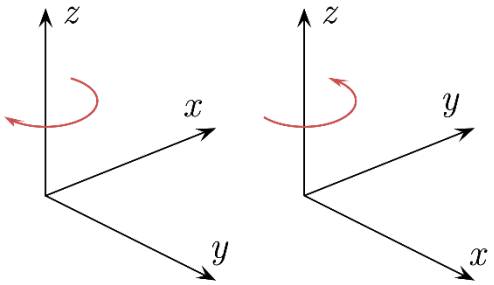

Illustration 20. Left- and right-hand coordinate systems.

Positive z, x and y coordinates of Left-hand System point respectively forward, right and up, furthermore the positive rotation is clockwise about the axis. However within the second system the sign of z coordinate is opposite and the rotation is counterclockwise.

2.7.3.2. Vector calculations

Vector describes a point and its position on the coordinate system. There are certain operations that can be performed on vectors:

- Vector length
- Normalization
- Dot product
- Cross product

Vectors with length of 1 (called unit vectors) are responsible for specifying the direction and movement of light. Normalization is a process of transforming any vector to unit vector. Normalized vector is denoted $\hat{x}$ and given by an equation:

$$\hat{\boldsymbol{x}} \equiv \frac{\boldsymbol{x}}{|\boldsymbol{x}|} \tag{1}$$

A length of a vector however is obtained by using a following formula:

$$||\boldsymbol{x}|| = \sqrt{\boldsymbol{x_1^2} + \boldsymbol{x_2^2} + \cdots + \boldsymbol{x_n^2}} \tag{2}$$

Next crucial operation is called the dot product. One of the approaches that can be used for the calculation is:

$$\boldsymbol{a} \cdot \boldsymbol{b} = ||\boldsymbol{a}||\ ||\boldsymbol{b}|| \cdot \boldsymbol{cos}\ \boldsymbol{\alpha}$$

(1)

Dot product is often used in light computations and sophisticated artificial intelligence calculations. Finally, cross product is an operation particularly useful when operating on polygons. For example it can be applied to obtain normal of polygons. It is calculated using the following equation:

$$\boldsymbol{a} \times \boldsymbol{b} = \boldsymbol{n}\, ||\boldsymbol{a}||\, ||\boldsymbol{b}|| \cdot \sin \boldsymbol{\alpha}$$

(4)

Together the two vectors describe a plane and the output is vector that is perpendicular to that plane. Finally it is important to apply the equation in a particular order as changing the order affect the sign of the result:

$$\boldsymbol{a} \times \boldsymbol{b} = -(\boldsymbol{b} \times \boldsymbol{a})$$

(5)

2.7.3.3. Matrices

Matrices are a fundamental concept in real time rendering and understanding of matrices is necessary to 3D game developers. Matrices are utilized when an object needs to be translated, scaled, rotated or moved.

2.7.3.3.1. Matrix translation

Translation in game development is a process of moving an object from one location to another. In order to simplify complex calculations a translation is assumed to take place in homogeneous coordinates[50] introduced by August Ferdinand Möbius[51].Translation for homogeneous coordinates may be computed within a single calculation and is given by matrix:

$$T(tx, ty, tz) = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[50] Homogeneous coordinates – system for representing coordinates of points, including infinity.
[51] August Ferdinand Möbius (1790 -1868) - German mathematician and astronomer

It is also possible to perform translation by a given point (x, y, z):

$$\begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + tx \\ y + ty \\ z + tz \\ 1 \end{bmatrix}$$

2.7.3.3.2. Matrix scaling

Scaling is a linear transformation modifying objects size by a scale factor. In order to scale an object by a vector, a point needs to be multiplied by a scaling matrix.

$$Sv = \begin{bmatrix} vx & 0 & 0 \\ 0 & vy & 0 \\ 0 & 0 & vz \end{bmatrix}$$

$$SvP = \begin{bmatrix} vx & 0 & 0 \\ 0 & vy & 0 \\ 0 & 0 & vz \end{bmatrix} \begin{bmatrix} px \\ py \\ pz \end{bmatrix} = \begin{bmatrix} vxpx \\ vypy \\ vzpz \end{bmatrix}$$

2.7.3.3.3. Matrix rotation

Rotation is used for rotating objects in Euclidean space. In order to achieve full rotation of a 3D object, three one-axis rotation matrices have to be multiplied in specific order. Firstly, a rotation along the Y axis has to be performed:

$$\begin{bmatrix} \cos \propto & 0 & \sin \propto & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \propto & 0 & \cos \propto & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Secondly, a rotation along the X axis has to be applied.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \propto & -\sin \propto & 0 \\ 0 & \sin \propto & \cos \propto & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Finally, a rotation along the Z axis has to be applied

$$\begin{bmatrix} \cos \propto & -\sin \propto & 0 & 0 \\ \sin \propto & \cos \propto & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.7.3.3.4. View Space Matrix

A view space matrix serves the purpose of transforming objects from the object space to the view space. View space is a coordinate system relative to the camera. In order to create such matrix three vectors are required. First of the vectors represents the position of the camera. Second vector defines the direction, which the camera is facing. Finally, third vector pointing to the camera's top. It is also perpendicular to the second vector.

$$\begin{bmatrix} xAxis.x & yAxis.x & zAxis.x & 0 \\ xAxis.y & yAxis.y & zAxis.y & 0 \\ xAxis.z & yAxis.z & zAxis.z & 0 \\ -(xAxis.x * v1) & -(yAxis.x * v1) & -(zAxis.x * v1) & 1 \end{bmatrix}$$

Where:

$$zAxis = normal(v2 - v1)$$

$$xAxis = normal(v3 * zAxis)$$

$$yAxis = normal(zAxis * xAxis)$$

2.8. Shaders

2.8.1. Introduction

Shaders are the key to improving the output of all kinds of virtual displayable media; from cartoons, movies and realistic artworks to computer games. The term "shader" was derived by a company named Pixar in May, 1998, and from that moment numerous major graphics software libraries such as Direct3D or OpenGL adapted Shaders.

They are utilized for tasks such as: production of appropriate levels of color within an image or production of post-processing effects. The rendering effect is calculated by code written to run on a GPU. Number of parameters like position, saturation, hue, contrast and brightness may be dynamically altered in order to produce a final image.

2.8.2. Shader types

Shaders may be differentiated into three programmable stages:

- Vertex Shaders
- Geometry Shaders
- Pixel Shaders

Vertex Shaders process each of the vertices from the input assembler[52] given to the graphics processor. They perform all the per-vertex operations and hold responsibility for transforming 3D vertex's position into 2D coordinates, displayed on a screen. Vertex Shaders handle input vertexes one at the time, returning a single output vertex for each of them. The number executions of the stage can be queried from the CPU. Furthermore, this stage must be active in order for the pipeline[53] to execute.

Geometry Shaders, unlike Vertex Shaders do not have to operate on a single input. They are capable of generating multiple graphic primitives like points, triangles and lines, based on the data sent to the pipeline. When active, Geometry Shaders are invoked once per every primitive, regardless whether it was generated or passed down in the pipeline. Typical uses of Geometry Shaders, among many others include geometry tessellation and sprite generation.

Pixel Shaders also known as fragment Shaders enable so called rich shading techniques and hold responsibility for computation of pixel attributes. In their basic form they output a single pixel on a screen, however in non-trivial cases are capable of handling multiple pixels. Pixel Shaders are also capable of applying shadows and light values. It is worth mentioning that only the Pixel Shaders can be used as a filters for video streams [Ebert98]

2.8.3. Shader Model 5.0

New shader model was introduced alongside with the release of Windows 7. Being a new generation, it however still is capable of runs on Windows Vista and supports DirectX 9. Contrary to previous upgrades, it was introduced to solve existing programming problems in game engines, instead of providing additional features.

In order to fully comprehend Shader Model 5.0, one must understand a serious bottleneck of rendering algorithms, namely the transfer of highly refined meshes between CPU and the GPU. Without ability to manipulate on primitives it is merely possible to implement any of the rendering algorithms in the GPU, thus leaving the mesh refining process to the CPU. Such a delegation of responsibilities puts a great strain on the bandwidth between Graphics Card and Processing Unit, hence affecting the performance thoroughly.

Shader Model 5.0 introduces Tessellation Stage to the pipeline. With the use of HLSL[55] in the Tessellation stage, programmers can directly access GPU generated triangle’s vertices coordinates,

[52] Input assembly – first stage of the graphics pipeline, responsible for reading and parsing primitive data provided by the user
[53] Pipeline – programmable pipeline created for generating graphics in real-time applications
[55] High Level Shading Language (HLSL) – high level shading language developed by Microsoft to facilitate Direct3D

thus trading the CPU-GPU bandwidth for the direct GPU operations that hardly affect the performance. Saving memory allows for rendering of higher detailed surfaces.

2.9. Particle systems

2.9.1. Introduction

Nowadays, modern computer games tend to rely heavily on graphical effects and overall quality of visuals. Especially in dynamic, action-driven games it seems desirable not only to render attractive static environment, but also to deliver convincing dynamic effects such as explosions and other complex, vigorous phenomena. While traditional mesh rendering methods may be utilized, a typical approach to modeling these chaotic visuals is to take advantage of a particle system.

2.9.2. Design

Typically, a particle system consist of a large number of meshes or sprites[56]. Elements in such a system tend to be rendered and disposed of quickly and should be organized in such a manner, as if to simulate a particular dynamic and somewhat fuzzy object [Reeves83], as opposed to 'solid' ones modeling terrain, buildings and other static entities. The lifetime of elements in a particle system may be divided into several phases:

- creation
- movement
- decay

A particle system requires the definition of a region in space representing a source of creation of particles. This is referred to as emitter. The emitter's position and shape determine the direction, scope of emission and overall behavior of particles.

The movement phase comprises the dynamic flow and evolution of elements, depending on each particle's individual set of variables. Among these, it is most common to specify the speed of emission, reaction to gravity and static environment (often involving collision detection) as well as their lifetime. In order to achieve a higher level of realism, all or some of these factors are randomized. For example, it is common for some particles to dissolve quicker than others, or change their color throughout emission.

Upon reaching its initial lifetime value, a particle is being disposed of. Alternatively, based on design, it may also be destroyed when colliding with a specific object, or even when a particular value has exceeded a given threshold. For instance, we would expect some light-simulating particles

[56] Sprite – an image representing a character or other object in 2D computer games

to dissolve naturally when fading out or increasing in brightness above a given level. It is crucial to choose these conditions carefully, as it is possible to create an overflow in the number of particles rendered at the same time, thus leading to decreased performance and consequently, significantly worse player experience.

2.10. Collision detection

2.10.1. Introduction

The vast majority of contemporary action computer games involve dynamic movement of meshes. In-game objects need to interact with each other in particular way, most often in order to simulate their real-world behavior. For example, in a racing game we would expect a car slow down and bounce off dramatically upon hitting a concrete barrier, or a bullet to inflict a damage if an enemy has been hit in a shooter-style game. It is therefore required to measure intersections of objects and fire appropriate events if such a collision has been detected. The correctness and efficiency of such measurements play a crucial role in providing players with desired experience, thus various methods of collision detection have been introduced to modern game engines. The typical approaches [Ericson05] involve creating components representing boxes, spheres with combinations of these as well as complete meshes.

2.10.2. Bounding boxes

The most straight-forward, and perhaps most naive approach is to approximate objects using boxes. A collision occurs when the boxes' boundaries intersect, meaning a calculations and comparisons of the cuboids' minimum and maximum coordinates are needed. This, however, holds true only if the boxes are aligned to a single axis. In the case of one object being rotated about a different axis, a collision may not be detected. A simple workaround involves approximating objects into world axes-aligned boxes (Axis Aligned Bounding Boxes), translating independent of the objects' rotation.

2.10.3. Bounding spheres

An even simpler collision detection mechanism involves the use of bounding spheres, in which objects' boundaries are approximated by spheres. A collision occurs when these spheres overlap, i.e. when the distance between the spheres' centers is smaller than the sum of their radiuses. Benefits of this approach include great simplicity of calculations, as only two values are needed to be compared in order to measure intersection. Contrary to the use of bounding boxes, sphere-based detection does not require normalization of the coordinate system, as spheres are described irrespective of alignment of the axes. On the other hand, it is unlikely that a computer game will

contain only round, spherical meshes, thus approximation of more complex shapes proves highly inaccurate, resulting in faulty collisions.

2.10.4. Mesh-to-mesh collisions

Mesh-to-mesh collision detection provides the most accurate way of measuring intersection between objects. This approach assures that meshes did intersect, as all polygons' edges of two objects are compared against each other, yielding calculations proving or disproving actual collision. Although exact, this approach proves inefficient due to its immense complexity, especially in the case of high-polygon meshes. Due to its possible slowness, mesh-to-mesh strategy is used only when collision needs to be measured with perfect precision.

2.10.5. Collider combinations

A simple and efficient way of increasing collision precision without resorting to full-on mesh-to-mesh collision is to combine two or more primitive-shaped colliders on a single complex mesh. For instance, a 3D model of a human will typically be approximated by several disjoint box colliders covering the character's arms, legs and torso, while a sphere collider may be positioned around head. While still prone for mistakenly measured collisions, this strategy offers a far better efficiency in comparison with mesh-to-mesh collisions and has been successfully utilized especially in quick-paced first-person shooter games such as Counter-Strike: Global Offensive (2012) [CSGlobal]. Additionally, these independent colliders function as hitboxes which simulate the character's body parts, yielding in greater damage when hit on the head's hitbox, and smaller in the case of shot reaching other area.

2.11. Artificial intelligence

2.11.1. Introduction

'Artificial intelligence' is a term which has been circulating academia and design laboratories since the introduction of earliest computing machines. The general struggle to build intelligent programs eventually manifested itself in the first computer games of 1970s. Typical examples include Pong (1972) played in single-player mode, in which the program is responsible for moving the opposing player's racket, based on the prediction of the ball's position. Consequently, it has to be noted that even the most primitive games demonstrated the need to provide a somehow realistic experience in order to keep the player immersed.

Vast majority of modern computer games involving dynamic gameplay[57] rely heavily on

[57] Gameplay – the overall flow and pattern of how a particular game was designed to be interacted with.

interaction of the player character with other entities in the virtual world. These may be stationary enemies, moving obstacles, or intricately designed NPCs[58] with complex state-machine-based behavior scenarios. It is therefore necessary to pay special attention to the implementation of artificial intelligence, as in many cases it appears to be the foundation of a playable and appropriately challenging experience.

2.11.2. Navigation and pathfinding

The ability of non-player characters to traverse the virtual scene is one of the most common challenges in the area of computer games AI. A naive workaround of this problem is to programmatically assign fixed movement paths to every character, as done in early platform games such as Super Mario Bros (1985). Although extremely easy to implement, this technique usually results in highly predictable and unnatural behavior not desired in modern productions, despite having been successfully introduced in vast number of simple arcade games of the 1980s.

Various methods have been introduced in order to simulate real-world movement of NPCs in obstacle-filled environment. The A* algorithm [Hart68] proves to be one of the most popular and efficient solutions solving the pathfinding issue. According to the general version of the algorithm, the first step is to divide the space into equally-sizes segments representing movable and non-movable space, the latter usually consisting of walls and other obstacles. After approximating the environment using tiles, a starting and finishing segments need to be determined. In each step a movement is determined based on the constantly updated list of available neighboring tiles (open list), along with another list containing information on already visited tiles (closed list). The process of choosing the next segment is based on the score given to every tile. The value is calculated as follows:

$$f = g + h$$

where g is the cost of moving from the starting point to the current one, while h represents the estimation of movement from the currently visited tile to the destination. The cost may represent an arbitrary concept derived from the field of optimization programming, but typically stands for the number of segments between appropriate segments dividing the virtual space.

2.11.3. Finite state machines

In order to ensure the fact that the non-player characters act in a realistic and convincing way, it is inevitable to account for more than one type of behaviors, depending on the in-game circumstances or certain events happening. A typical example of this strategy is connected to the problem of modeling the behavior of enemies, who may be expected to attack the playable character

[58] NPC (Non-player character) – an entity in the game not controlled by the player.

only when he/she has been spotted directly, or simply appeared within particular distance. In such a case, a concept of a state machine appears to be particularly well-suited.

State machine is an abstract idea used to describe and model a system's behavior based on its current state. A finite state machine may be in only one of a finite number of states. A transition from one state to the other may happen if certain event has been fired or a particular logical condition has been met. A single state machine may be easily reusable, as in most cases in-game characters tend to be divided into groups sharing a common role pattern. States and transitions are therefore simple and powerful means of defining complex behavior scenarios of NPCs.

Technically, a properly designed finite state machine must definite a complete list of all possible states of a given agent, a list of all possible transitions between states, a list of logical rules pointing to a transition and a list of all possible external events triggering a transition. Additionally, this definition requires the choice of a single initial state which will then develop into another through transitions. For instance, a typical state-based system for a guard may include states and transitions associated with moving around the room, chasing the player and engaging in combat, as well as being destroyed by the player (*see Illustration 21*).

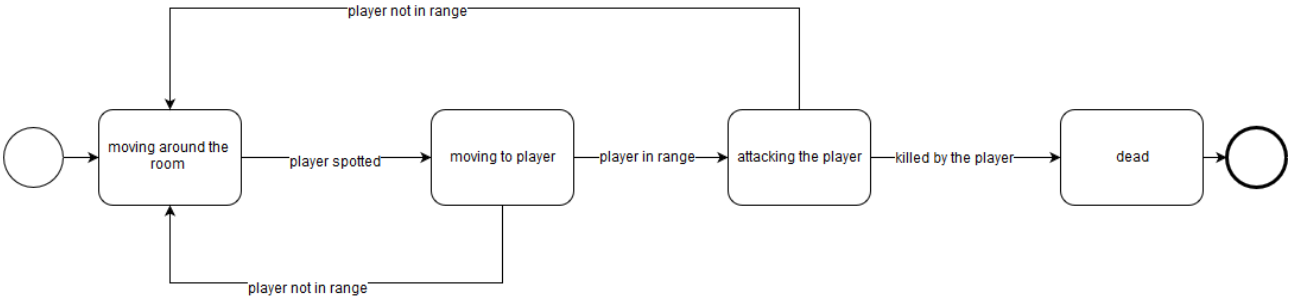

Illustration 21. Sample state diagram for an enemy character.

It has to be noted that this is a rather simplistic, albeit common and efficient technique to determine state-based behavior of AI-equipped characters. Contrary to this fully deterministic approach, there exists a concept of a non-deterministic finite state machine in which transitions cannot be clearly resolved if current state is known. In such a case, fuzzy logic needs to be involved, as changes from one state to the other depend on a set of multiple events happening simultaneously, usually with varying intensity, thus making it impossible to clearly distinguish the next state. The implementation of a non-deterministic finite state machines involves assigning weight values to all the inputs, as well as a threshold value deciding if an input has to be considered when evaluating possible transition to another state. A possible extension of this mechanism could include means of input weight randomization in order to achieve even more realistic behavior thanks to the introduction of an element of unpredictability.

3. Design and implementation section

3.1. Design

3.1.1. Game design

As in the case of most averagely complex IT solutions involving the creation of software from the very beginning, a precise plan had to be prepared. Although the general idea of what "Laboratory Night" should look like had already been known, it turned out to be necessary to compile the list encompassing the entire project's scope in order to keep the work organized and enforce particular routines during actual implementation.

Contemporary computer games market is a big and mature one, resulting in vast numbers of hugely diverse and innovative products developed and released each year, yet most of them clearly sticking to one of well-known genres. It has thus been obvious that the project must be easily identifiable as belonging to one of these types, as well as comprise some features that are somewhat exotic to the common understanding of a given genre. After a series of discussions and brainstorming sessions, a decision has been made to create a shooter-type game. Next, a story outline had been written down in order to provide basic information on the project's setting, environment and style of gameplay. Eventually, a list of all desired features and mechanics had been prepared, which was soon stripped down in order to fit the given 4-months deadline period. Finally, after having applied more corrections, a complete design document was compiled, containing precise information on the game's story background, visuals, mechanics and the player's ultimate goals.

The plan assumed the game to be set up in a dark, science-fiction-type world in which the player-controlled protagonist – a futuristic soldier-scientist – has been trapped in a laboratory as a result of a failed experiment on the nature of gravity. The player must make his/her way through and out of the building, having to fight hordes of revolted robots and rogue lab technicians, eventually trying to defeat a gigantic humanoid robot – the main antagonist. As a result, the scientist is required to make use of a range weapons including a standard rifle, a shotgun and a laser emitter. The two latter ones were not supposed to be available from the start, as they had been intentionally designed to be much more effective than the regular gun.

Moreover, it was decided from the start that the player will be able to use more than the weapons above, which are rather typical to the shooter game genre. This is where the concept of a Gravity Gun came to be. Taking advantage of the experiment's consequences, the protagonist has been equipped with a prototype of a weapon which enables grabbing various movable objects, even on a long range. As a result, the player was provided with another strategy of fighting and tricking enemies. More importantly, the introduction of a Gravity Gun resulted in major reconsideration of

the game's design by including certain scenarios in which the use of a new weapon would be necessary. One of them is when a rocket-launching turret is encountered. The only way to destroy it is to catch an incoming rocket and shoot it back at the enemy, thus forcing the player to abandon the usual 'shoot'em up'[95] strategy. Additionally, it was decided that attackers hit by objects thrown via scientist's Gravity Gun would receive additional damage in order to promote the unique feature and encourage the player to utilize unconventional methods of fighting enemies.

Finally, in order to enhance and diversify the gameplay, the scientist was given an ability to activate special gravity-induced skills, similar to these found in certain fantasy and science-fiction-type games. These include:

- Shield used to protect the scientist from enemies' bullets
- Black hole drawing nearby enemies and movable objects in order to be easily destroyed
- Shockwave being a powerful long-range attack destroying every hostile on its way**.**

3.1.2. Team and choice of technology

From the very beginning, "Laboratory Night" was intended to be completed within a 4-month period by a 2-person team with no experience in game development. Considering these assumptions, it was decided that an already available game engine should be used. While it is common for some major game development studios focusing on AAA games to create their proprietary solutions in the Microsoft DirectX environment, the scope and restrictions of the project required usage of a mature, effective, quick-development tool. Eventually, the Unity engine was selected as the technology of choice.

The Unity engine proved to be a powerful and relatively simple tool. The prime and most obvious advantage is the utilization of C# programming language, which the authors had been familiar with prior to the beginning of "Laboratory Night" development. Contrary to hardware-near DirectX API written for the C++ language, the high-level[96] Unity technology running C# turned out to be a simple, yet effective technology yielding satisfactory results in relatively short time.

Furthermore, the existence of the Unity Asset Store proved extremely helpful in the process of collection of various assets featured in the project. Vast number of both free and paid 3D models, textures and audio files are – among others – easily available on the Asset Store and may be included in the game in accordance to the Store's license. As none of the authors had been familiar with 3D modeling or texturing tools on a professional level, it was decided that ready-made visual assets will be downloaded from the Asset Store and used in the project in order to speed up the developed process and focus on creating and polishing the gameplay.

[95] Shoot'em up – a video games genre basing their gameplay on shooting enemies.
[98] Debugging – a process of detecting and eliminating faulty software behavior.

Yet another decision had to be made during the design and analysis phases, as it was during that time that a new version of Unity engine was released. The newly released version 5 was made public when the older version 4 had already been well-established in terms of documentation and other references. However, it was quickly decided that the newer version will be used thanks to a variety of major improvements which would speed up and simplify the development process as well as result in a product of a higher visual quality. These enhancements include a new lighting technology, the introduction of a standardized, cross-platform shader and compliance with PhysX 3.3 specification advancing the built-in 3D physics system [UnityScripting].

Furthermore, as the majority of work on the project would ultimately involve writing source code, an IDE had to be decided on. Microsoft Visual Studio 13 was chosen as a result of its extensive debugging[98], refactoring[99] and autocomplete[100] features.

Finally, as the project was intended to be developed by a 2-person team, the Git version control system was selected due to its distributed, decentralized nature and freely available repository hosting services such as GitHub[101] or Bitbucket[102]. Specifically, the used version control system stores history in which every collaborator's input is easily distinguishable and can be restored at any time, making the development process much safer and simpler (*see Illustration 22).*

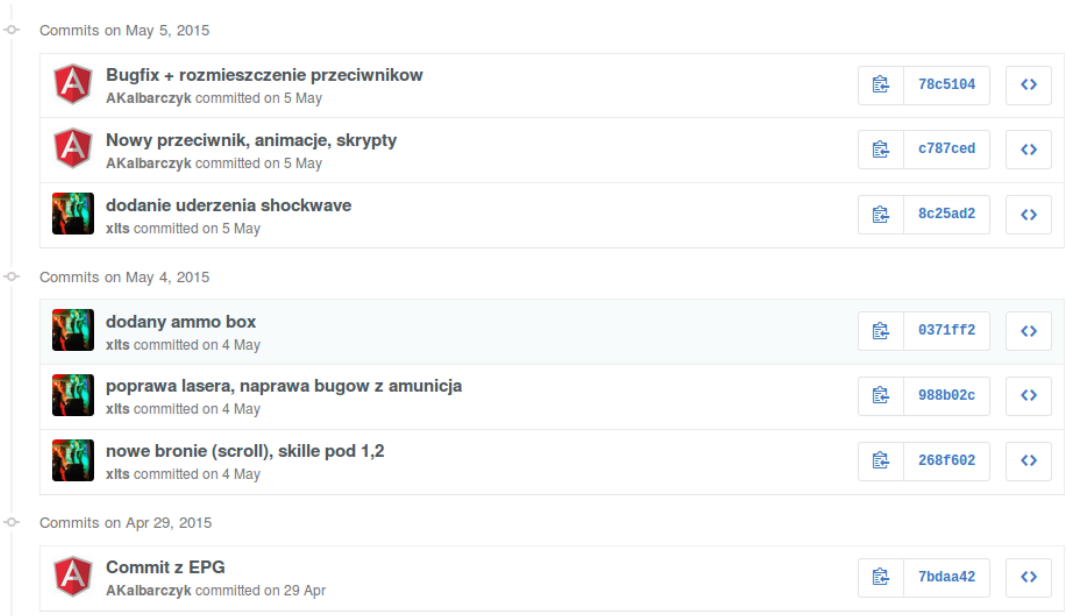

Illustration 22. Commit history in a Git version control system.

[98] Debugging – a process of detecting and eliminating faulty software behavior.
[99] Refactoring – a process of redesigning and restructuring source code in order to achieve better organization and readability.
[100] Autocomplete – an IDE functionality that provides automated suggestions when writing source code.
[101] GitHub – a website used to freely host source code files in public repositories.
[102] Bitbucket – a website used to freely host source code files in private repositories.

3.1.3. Project and source code organization

From the moment the list of desired features had been compiled, it was obvious that the project's source code will be lengthy and expand beyond what can be observed in various sample or tutorial projects. Therefore, a certain structure and conventions had to be enforced in order for the project to remain organized and readable. As in the case of moderately- and large-sized software, special attention had to be paid to appropriate isolation and encapsulation of scripts and classes due to the assumption that a single component should remain easily modifiable without the necessity to change other, dependent ones.

Although the Unity engine usually does not enforce particular organizational conventions, a basic folder structure is established upon creation of a new project. It was initially decided to stick to this given format, as it often referred to in the documentation and official manuals. Consequently, all project-specific elements have been stored in the generated Assets directory. Further expanding the structure, the folder contained more sub-directories, as listed below:

- Scenes containing the Unity .scene files used in the game
- Materials containing Unity .mat files to be used on renderers
- Models grouping 3D models
- Prefabs containing ready-to-use Unity GameObjects to be put in a scene
- Resources containing some of the prefabs in the above directory. This exact folder name is required in order to programmatically load and render assets during the program's runtime.
- Scripts containing all the source code files
- Shaders containing custom shader definitions
- Textures containing all the texture files

As the number of assets began to grow rapidly throughout the project's development, this self-imposed structure became critical in terms of being able to add, remove and modify the files conveniently.

Because most visual assets have been provided via the Unity Asset Store, much of the attention has been brought to writing source code files. Nowadays, the clarity and expressiveness of source code is universally considered crucial to keeping the code base maintainable [Martin08]. This is why yet another conventions needed to be defined. As C# had been chosen as the development technology, most of the language's recommended best practices were adopted.

In particular, interface, enumeration, class, as well as public member names were written in the Pascal Case notation[103], while function parameters and local variables were named in Camel

[103] Pascal Case notation – a name-formatting convention in which every word starts with a capital letter.

Case[104] style. It was of most importance that all the code-specific names should be as expressive as possible, resulting in sometimes long and complex method, variable and class signatures, yet unambiguously defining its purpose and behavior. For example, a variable could easily be named "inputX" or "inputHorizontal", but only the latter option provides full description of its functionality.

Classes were organized with as much compliance with the Single Responsibility Principle[105] [Martin02] as possible. In order to retain the code maintainability and modularity it was necessary to translate desired functionalities into classes responsible for small and independent actions. For example, the class PlayerHealthController only held information on the player's health value and its purpose was to add to and subtract from this number as well as compare it to another parameters. Other operations, such as setting it on a GUI health bar or checking if a health pack has been collected were delegated to appropriate classes: GUIBarScript and AmmoBoxController, respectively.

3.2. Implementation

3.2.1. Controls

Being a dynamic, action-driven shooter-style game, the control of "Laboratory Night's" main character had to be carefully designed, as much of the player's experience would rely on smooth and responsive input. As early as in the analysis phase, it became clear that correct implementation of scientist's controls will most likely exceed what is seen in typical shooter games.

In order to provide desired user experience, it was decided that much of the player's input will be implemented according to certain standards existing among games in similar genres. Intuitively, the player's character could be rotated by moving a mouse. Furthermore, the scientist can be moved either by pressing W, A, S, D or up arrow, left arrow, down arrow, right arrow, respectively. Moreover, for further possible development, it was crucial that the implementation of any character movement was to be made independent from keyboard-specific keys. Instead of explicitly associating a single key with particular action, it was enough to utilize a much more generic approach that would fetch a value representing virtual horizontal or vertical axes which are then translated into actual key mapping specified in platform-specific options which could differ significantly.

As much of "Laboratory Night" mechanics center around shooting, the weapon and special skills controls had to be implemented with special care. A special module describes all user input associated with shooting. As typical for similar games, the Left Mouse Button serves as means of using a basic weapon, while Mouse Scroll is responsible for switching weapons, if they are currently

[104] Camel Case notation – a name-formatting convention in which the first word starts with a lowercase letter and the rest start with capital letters.
[105] Single Responsibility Principle – a design pattern saying that each module in a software system should perform only one function.

available. Right Mouse Button, usually functioning as an 'alternative fire' control, launches the Gravity Gun and 1, 2 and 3 buttons activate the scientist's special skills: Shield, Black Hole and Shockwave, respectively. Although the controls may seem complicated at first glance, most players would be able to accustom themselves quickly thanks to their intuitiveness.

Finally, the camera was also designed to stick to conventions seen in other games with 'top-down' view. Therefore, the character is followed at all times and the view rotation is fixed, allowing for a clear perspective on the scene.

3.2.2. Basic gameplay logic

3.2.2.1. The player's quest

Although conceived as a shooter game, destroying enemies is not the game's ultimate goal *per se*. Ultimately, the player finishes the game upon reaching the final room, thus escaping the laboratory. Therefore, some interactive, quest-related objects were created:

- Hackable computers
- Unlockable doors

The process of moving through the areas focuses on finding hidden computers and hacking them in order to unlock doors leading to different spaces. These are painted in matching colors so that it is obvious that, for instance, hacking a blue-colored machine opens blue-colored doors. Finally, in order to minimize the players' confusion, the gates were positioned in such places as to ensure that the player does not see the machine before having noticed a specially marked closed door.

3.2.2.2. Combat

From the start, "Laboratory Night" was intended to be an action-shooter game, therefore the major part of the gameplay is centered on firing weapons. Generally, the protagonist has no choice but to fight off waves of enemies in order to proceed to further rooms and eventually escape the laboratory. As in other games in the genre, the player has a choice of several weapons, provided he was able to find and collect them throughout the play. Another way to destroy an enemy is to launch a movable object with the use of a Gravity Gun, or by activating a powerful Shockwave special skill.

In order to properly balance the vastness and diversity of player's weaponry, fighting hostiles needed to appropriately challenging. Upon noticing the character, an enemy would instantly switch to an attack stance, trying to move close to the player and harm him. Consequently, not only does the player need to focus on aiming, but also keep on trying to avoid attackers' bullets. Special attention has been paid in order to make the combat energetic, and often chaotic. Hostiles fire at a high rate and are usually gathered in rather large groups. What is more, Gravity Gun or special skills are also available, thus a proper strategy is necessary.

3.2.3. Meshes

Unity allows for great interactivity with 3D models, however itself does not provide modelling tools. It is worth mentioning that it does not only support triangulated, but also quad angulated meshes. In "Laboratory Night", various types of mesh formats[106] were be accepted, imported and eventually converted to internal format of the engine.

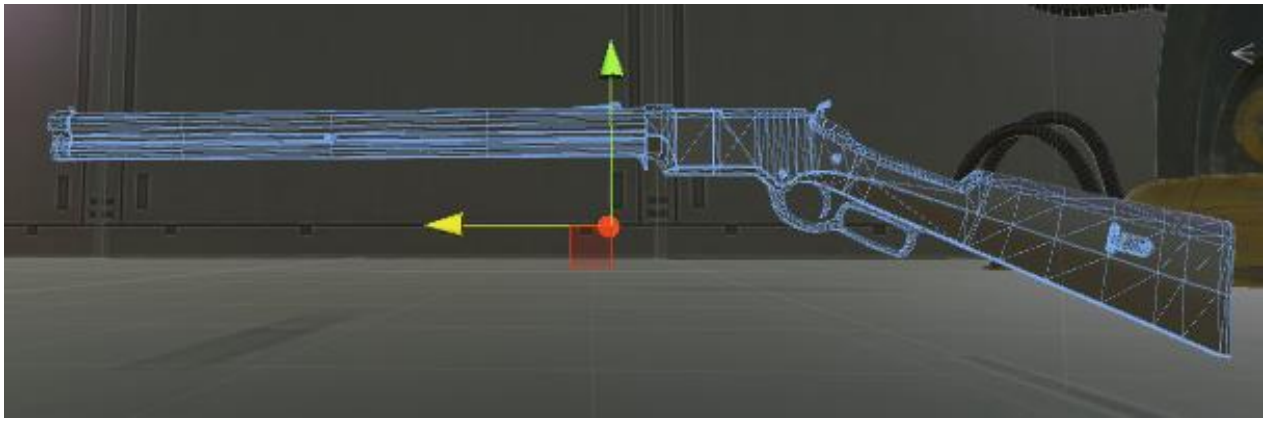

Illustration 23. Sample gun mesh from "Laboratory Night".

In order to maintain high FPS rate, "Laboratory Night" heavily depends on low poly models such as hand guns weld by player (*see Illustration 23*), containing meshes with small number of vertices.

However, due to importance of the main character, and in order to maximize user experience, a few high poly models, such as the scientist were also implemented in the game.

3.2.4. Texturing

3.2.4.1. Introduction

Taking advantage of possibilities offered by the Unity environment, the entire texturing process in "Laboratory Night" was handled by the engine's embedded technologies. Although mappings were fully automatized, all images and components had to be prepared and set up in order to achieve the desired results. All images used for texturing objects in "Laboratory Night" were taken from the public domain or downloaded from the Asset Store and included in the game based on an appropriate license.

3.2.4.2. Textures and materials

The texturing process in "Laboratory Night" involved preparation of appropriate bitmap files to be mapped on 3D models. Usually, the .tif file type was used, being a popular extension handled

[106] Mesh format – the game engine accepts numerous mesh file formats. Most important among them are: fbx, obj, dae and 3DS

by raster graphics tools. The files were stored in the Assets/Textures directory and automatically imported in order to be used as textures.

The process of preparing images to be rendered on an object started with the creation of a material being a special type of object representing a template of what the texture will look like once mapped to a mesh and rendered. Specifically, a single material is a data structure containing information on the texture, selected shader and its complementary attributes. Additionally, the material specified its albedo controlling the object's single, simplified primary color, the file serving as a normal map [UnityMaterials] specifying indentations and other details *(see Illustration 24)*, as well as other properties improving the level of detail or manipulating the object's presentation.

Illustration 24. A raw model (left), with applied texture (middle) and normal map (right).

All basic, repeating surfaces were set up to be slightly darkened and at the same time reflective in order to make the bright lighting feel more dramatic. On the other hand, enemies and turrets had their heightmaps attached and colors saturated to make them stand out from the environment and be easily identifiable by the player.

3.2.5. Lighting

Appropriate lighting is crucial helpful when building game's mood, especially in dark, science-fiction productions like "Laboratory Night". It is fundamental part of object shading and shadow casting. A typical lighting system needs certain properties such as direction, intensity and color in order to calculate shading of an object (*see Illustration 25*). In "Laboratory Night" different types of lights were utilized depending on the desired effect. These include point lights, spotlights, directional lights and area lights.

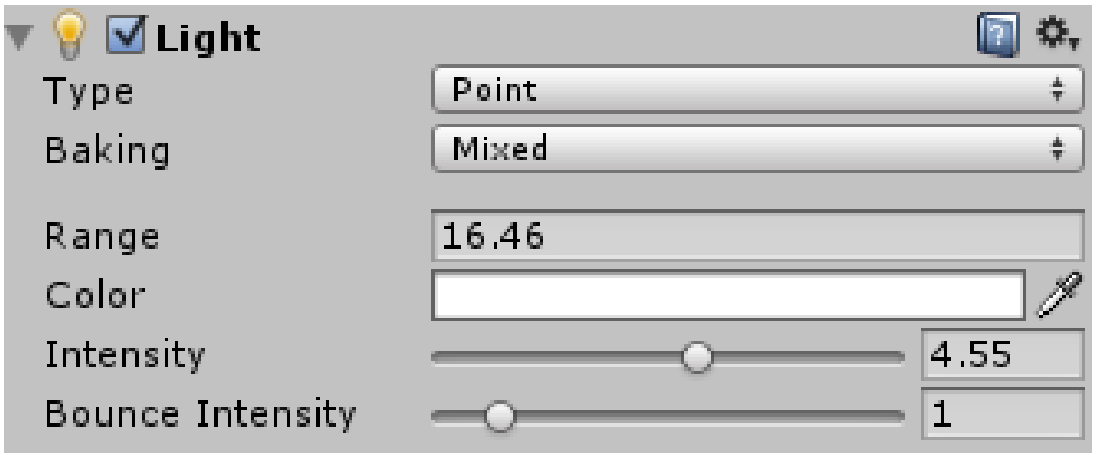

Illustration 25. Light object properties.

3.2.5.1. Point lights

Located in space at a certain point. Light is send in all directions equally. Its intensity diminishes as the distance from the source increases, eventually reaching zero (*see Illustration 26*). This type of light is was used for imitating various sources of light, such as lamps. Furthermore, its intensity was set to a high level in order to achieve an appropriate dynamic effect, while the purple color gave the surroundings an eerie, sci-fi feel.

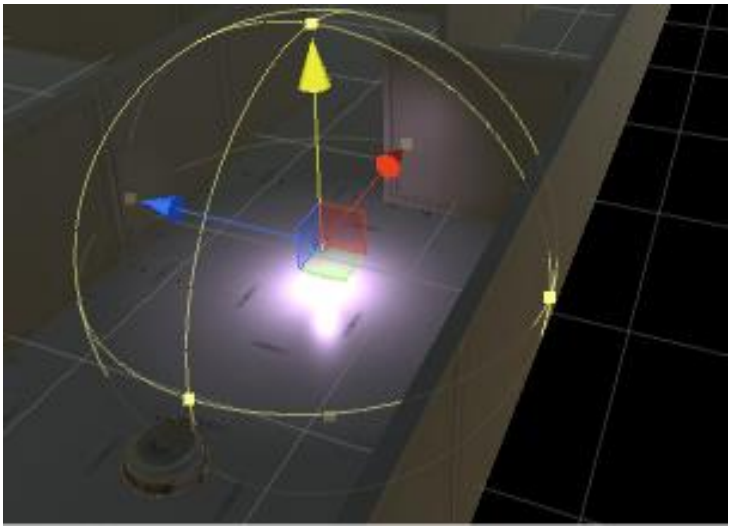

Illustration 26. Point light applied in "Laboratory Night".

3.2.5.2. Spot lights

Unlike Point light, this type of light is emitted only over a certain arc of space, resulting in cone-shaped region of illumination. Center of the cone indicates forward direction of the light (*see Illustration 27*). In the project, it is used to imitate strong, artificial light sources such as flashlights or headlights of a Floor Bot as well as to indicate teleports ending each level. Direction of light can be controlled using animations or scripts. It can also be attached to a moving object.

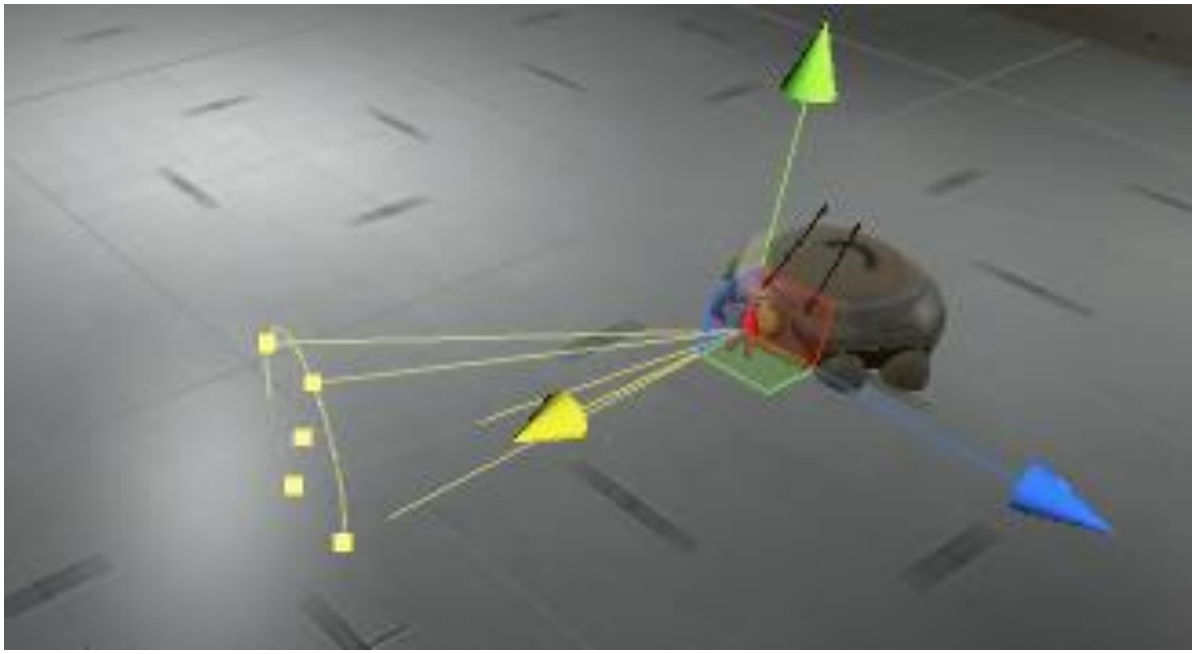

Illustration 27. Spot light applied in "Laboratory Night".

3.2.5.3. Directional light

Characteristic feature of directional light is that it does not have any identifiable source of emission, thus the light does not diminish with the increasing distance from the target *(see Illustration 28*). Usually it is used to represent a distant source, for example sun. In "Laboratory Night", a low-intensive directional light was used to delicately illuminate the scene and render global, one-directional shadows.

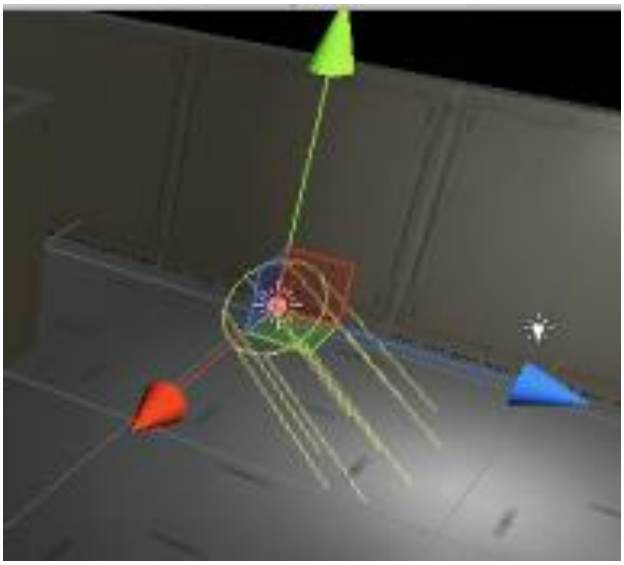

Illustration 28. Directional light in "Laboratory Night".

3.2.5.4. Area light

Area covered by this type of light is defined by a rectangle. Light is emitted only from one side of the rectangle, and falls over a specific range (*see Illustration 29*). For example it can be used

to simulate light emitted from buildings [Mathis15]. Due to the fact, that Area light is highly processor-demanding, it is not generated during the runtime. In “Laboratory Night”, it was used to simulate a global illumination effect to achieve a common basic luminosity level.

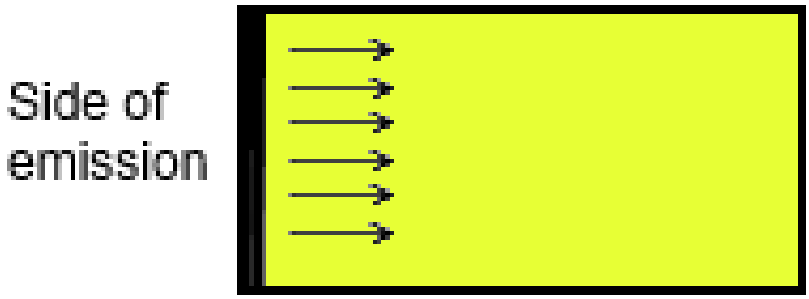

Illustration 29. Area light model.

3.2.6. Shadow mapping

The character of shadows in “Laboratory Night” is strongly dependent on the type of used lights. As point lights constitute the major part of illumination system, strong and contrasting shaded surfaces are rendered in real time using the shadow volumes method. Due to the large number of emission sources and a large area displayed on the screen, only hard shadows are generated in order to improve the graphics engine performance without a visible loss of quality.

Another feature introduced to speed up the rendering phase is the introduction of pre-generated static shaded areas. Using the lightmapping method, these were created based on the position and shape of stationary decorative objects in the scene that were never intended to be moving. As a result, calculations for these shadows do not have to performed every frame, relieving the GPU from much of the stress.

3.2.7. Rendering

Most available graphics engines allow developers to select from many rendering techniques. At the early stage of “Laboratory Night” development, one of the approaches had to be chosen. However, there were few factors of each of the techniques that had to be taken into the consideration. Two of the most common approaches were investigated: forward and deferred rendering.

Forward Rendering can be less hardware demanding, if certain conditions are met. Within this approach each of objects is rendered upon every intersection with light affecting it. On other words, number of times an object is rendered depends on the amount of light. Thus, the more the lights, the smaller the performance.

However, in Deferred Rendering, costs of rendering depends not on the number of intersecting ray, but it’s proportional to the number of pixels that the light illuminates. At most times this technique requires more powerful hardware.

A final assessment has been conducted that Forward Rendering will be an appropriate choice (*see Illustration 30*). "Laboratory Night" is a game with relatively small number of light source per scene, hence objects will not undergo the rendering process numerous times [UnityLighting].

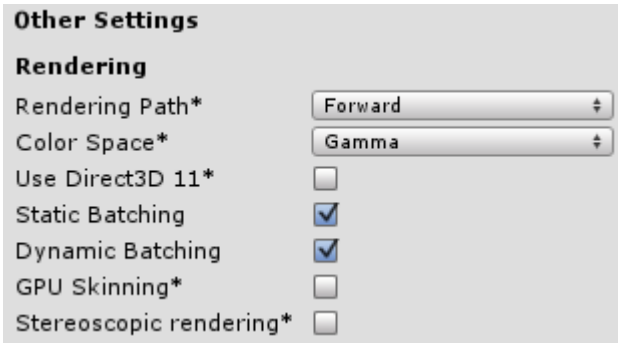

Illustration 30. Selection of a rendering technique.

3.2.8. Shaders

Typically, three different types of shaders are offered by graphics engines:

- Surface shaders
- Vertex and fragment shaders
- Fixed function shaders

However, regardless of selected type, every shader will eventually by wrapped in a language called ShaderLab, which is responsible for organization of shader structure.

"Laboratory Night" heavily depends on Surface shaders. The reason behind the choice, is that Surface shaders are rather templates for generating shaders. Some of the processes are done without developer's involvement, thus limiting the number of possible bugs [UnityGraphics]. In order to simplify and unify the material creation process, the newly-introduced standard shader was chosen to render the majority of "realistic" surfaces such as walls or pieces of furniture, especially due to its configurability and convincing visual performance.

In the case of "special" surfaces such as particle effects or quest-related doors, other dedicated scripts were used. In particular, shaders from the FX group helped to properly render various transparent and glowing surfaces. A special shader was prepared to be put on turret rockets in order to make them bright and distinctive. Finally, a special unlit shader proved extremely useful in the creation of a helper minimap which involved rendering the scene from an orthographic perspective with all lighting effects discarded.

3.2.9. Particle systems

Particle systems play a major role in most action-driven video games. Similarly, in the case of "Laboratory Night" it was necessary introduce convincing, visually appealing images of

explosions, shots and others, sometimes very subtle effects.

Most importantly, all in-game explosions are generated with the use of a particle system. The construction of a simple fireball rendered upon the bullet hitting an enemy involved utilization of a Particle System generation utility built in the Unity engine. Eventually, a ready-to-use prefab with proper specification was created. In particular, the explosion was designed to be a particle generator with a sphere-shaped emitter which would rapidly radiate with a high number of fiery-colored fragments with a very short lifetime in order to provide appropriate dynamics. This basic component served as an easily-customizable blueprint for creation of other, slightly varying types of explosions that would be fired when a rocket is destroyed or to improve the visual quality of enemies' bursting weapons.

Moreover, yet another type of a particle system had to be used in order to simulate the gravitational field induced by the scientist's Gravity Gun. In this case, a special prefab was selected, involving much lower fluctuation of elements. Furthermore, the effect duration was in fact matching exactly the pressing time of a right-mouse button, but the overall smaller emission rate compensated for possible optimization problems.

3.2.10. Collision detection

The implementation of a collision detection system is crucial to the creation of realistic gameplay in a shooter-style computer game. In the case of “Laboratory Night”, the implementation involved creating a basic, yet efficient solution based on simple shapes representing colliders.

From the start it was decided that no object requires a mesh collider component. This is because a dramatic drop in performance would be seen considering the fact that much of the processor's resources were dedicated to detecting intersections, resulting from the gameplay's overall character. Therefore, the first step constituted proper matching and configuration of components to the meshes' shapes in order to achieve best approximation. Typically, the player avatar as well as Mad Scientist enemies were given the capsule-shaped collider components in order to fit the armature's shape (*see Illustration 31*). The Floor Robot's round, compact model made it a candidate for the spherical collider components. This turned out to be especially beneficial computation-wise, as robots constituted the majority of the game's enemies, thus reducing the overall intersection detection complexity thanks to the use of a plain spherical collider. Finally, all walls, floors, doors and pieces of laboratory equipment were placed on the scene with box collider components attached.

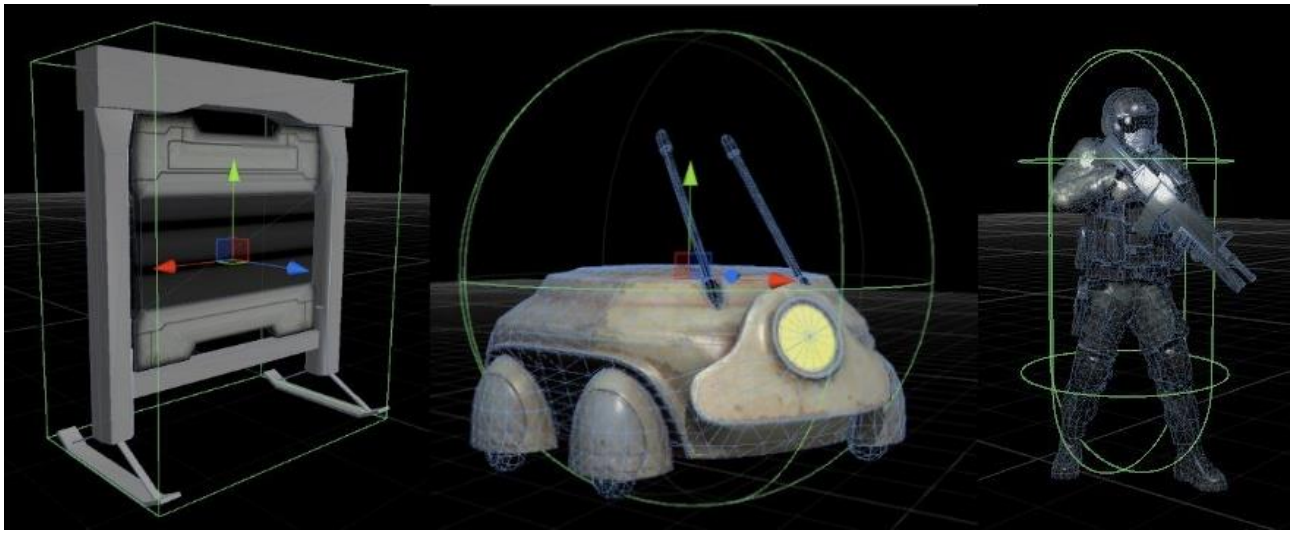

Illustration 31. In-game objects equipped with box (left), sphere (center) and capsule (right) colliders.

The implemented collision detection system was designed to work in close collaboration with the rigidbody[107] physics system. Specifically, if a collider-equipped object intersects with one rigidbody component, the latter will behave according to the rules of physics engine, even if no 'on-collision' action has been specified. Obviously, such behavior is not always desired, as custom, rigidbody-independent logic had to be applied. For instance, enemies hit by a movable object via a Gravity Gun were not intended to fly off chaotically with very high momentum. Instead, much less force was added in order for the attacker to remain functional and able to challenge the player.

As a result, precise definitions of actions to be triggered in the event of collision had to be written. What is more, handling of intersection events was oftentimes distributed over several modules so that various types of actions could be performed in the case of collision, in accordance with the Single Responsibility Principle. It is clearly visible when an attacker is hit by the scientist's bullet, where it is necessary to decrease the enemy's health value in the designated class, as well as create an explosion effect, which is a responsibility specific to the script describing behavior of a single bullet. Both procedures are run within the same trigger event, and technically could be put in a single collision-handling function.

Finally, plain physics-based collisions were designed to occur from hits of movable, gravity-induced objects. A designated module was created in order to define actions to be performed apart from the engine-specific rigidbody logic. Specifically, when a wooden box had been made to hit more than two enemies, it was supposed to shatter and not be usable anymore.

[107] Rigidbody system – a technology used to simulate realistic motion of in-game objects when interacting with other dynamic elements.

3.2.11. Artificial Intelligence

3.2.11.1. Introduction

Virtually all mechanics found in “Laboratory Night” revolve around full-on, dynamic combat. In order to provide desired experience, enemies had to be designed in order to behave in a realistic way and pose appropriate challenge. Therefore, proper implementation of their artificial intelligence proved necessary. In particular, the attackers needed to demonstrate certain pathfinding “abilities”, as well as to have certain easily noticeable behavior patterns. Achieving a desired effect involved the design of pathfinding and state-machine systems.

3.2.11.2. Pathfinding

Apart from the stationary rocket-launching turret, all enemies were intended to be mobile. As rooms would be filled with various pieces of equipment, furniture and other solid objects, the attackers were required to be able to find their way through these obstacles, trying to reach the player. Therefore certain means of pathfinding behavior had to be implemented.

The solution involved integration of several components interacting with each other that would comprise the universal pathfinding system. First, in the process called baking, it was necessary to define the area which the enemies could traverse (*see Illustration 32*). Only the floor was selected, as no entity had been designed to move on ceilings or walls. Next, the actual movement module had to be declared and put on appropriate characters which would try to reach a target being the game protagonist. Enemies were then programmed to get to the scientist omitting all obstacles in the form of pieces of furniture and static environment placed on the movable area. The navigation agents utilize a built-in A* algorithm implementation to find their way through these obstacles.

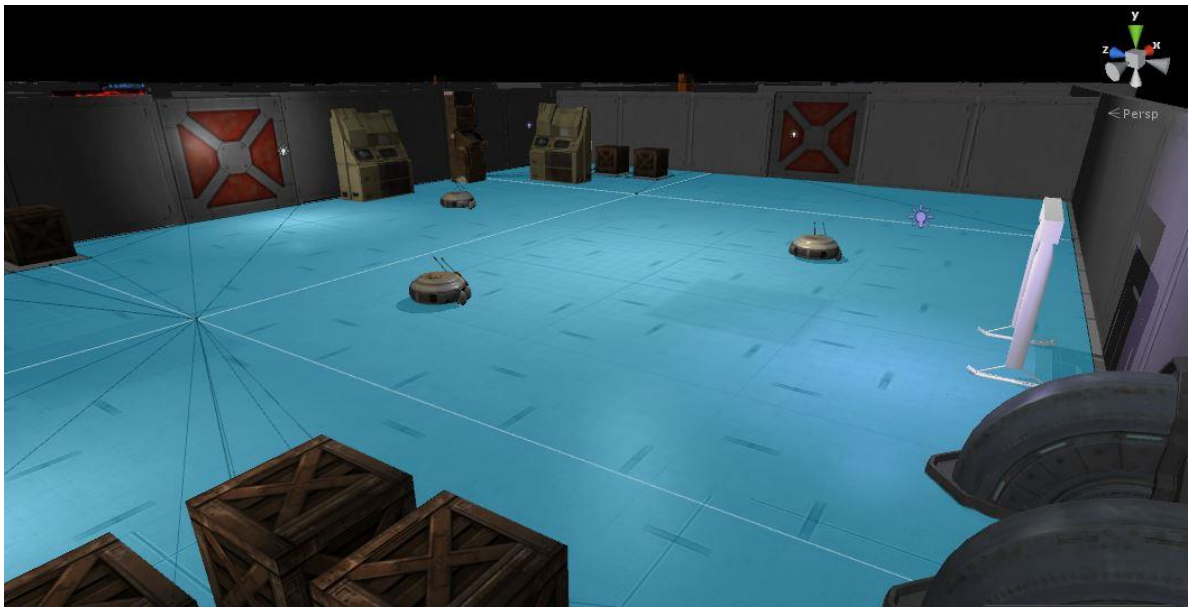

Illustration 32. Navigable area in a pathfinding system.

3.2.11.3. State machines

The usage of state machines in 'Laboratory Night' is somewhat limited. Very few states and transitions were defined in order to implement desired AI logic, thanks to the simplification of the pathfinding process. For example, it was no longer necessary to include a transition that would be fired when an enemy reached the maximum distance from the player. This issue was easily solved by specifying an appropriate stopping distance value on moving agents. Naturally, not every problem could be dealt with using built-in components, hence the existence of simple state machine-like elements.

All enemies on the level were designed to have two basic states: Idle and Hostile referring to the agent's behavior. The former, being an initial state, simply means that an attacker is stationary and does not use its available arms. The latter involved activation of pathfinding component, as well as a weapon. The only defined transition is fired upon the player appearing within an enemy's range given as a particular radius value. From this moment on, the attacker enters a Hostile state which remains active until it is destroyed by the player. This scenario is slightly different in the case of the turret AI. There, the transition is triggered only when the scientist crosses an explicitly defined boundary in the form of a spherical collider. When this area is left, the turret returns to the Idle state, provided it has not been destroyed before.

Another aspect requiring introduction of a custom state machine involved defining the story-based door-opening procedure. Having two states, Closed and Open, the transition happens when an appropriately colored computer is being hacked. As the concerns of scripts controlling doors and computer are clearly separate, one module serves as a listener and fires the transition only when a message from a computer-controlling module is received.

Finally, a flat state-machine-like structures help to handle meshes animations properly. Enemies switch animations when one of their state is changed. In the case of a player character it is slightly more complex, as different animations are loaded based on the player' input. These include Idle Aim motion when player is stationary, Walk when any of the movement keys is pressed, Walk Firing when both the mouse button and any of the movement keys are pressed, and Idle Firing when the character is stationary, but a mouse button has been pressed. The logic of this behavior is handled by the specific script holding a reference to a module responsible for handling the state machine.

3.2.12. Sounds

In order to provide a desired level of realism and player immersion, “Laboratory Night” required a substantial number of musical pieces and, especially, different environment sounds. Most basically, a single looping track as a scene’s ambient theme is played. These had to prepared to fit the atmosphere of the two levels as well as the menu screen.

Furthermore, the project required a great number of other tones representing the sounds of footsteps, various weapons, explosions and other special effects or noises. The handling of all sounds playback was delegated to a single instance of sound-controlling module which was designed to provide simple interface to play a single desired tone, according to performed action or a special event like shooting a rifle or using a skill. Finally, it has to be noted that all music and sounds used in "Laboratory Night" are works in the Public Domain.

4. Conclusion section

The goal of this thesis was to describe the entire process of creating a computer game, based on the example of the project “Laboratory Night”. Focusing not only on the necessary technical details, this work’s intention was also to illustrate the back side of the development stage, including planning and organizational aspects.

4.1. Final overview

As a result, a small yet functional video game was created, featuring some of the most important components found in commercial products of a larger scale. In terms of graphics, “Laboratory Night” can be compared to arcade shooter games available on many platforms, thanks to the engine’s support of high-quality dynamic lighting, advanced shaders and particle systems. Furthermore, complete collision detection and navigation systems were implemented, while state-based solution served as means to drive the behavior of non-player characters. More importantly, the project features a functional physics system on which the majority of mechanics was built on, making it stand out from other similar productions on the market.

“Laboratory Night” was successfully completed after an intensive 4-months development period, mainly due to careful planning and analysis performed beforehand. The creation of a detailed design document and a schedule helped the authors to establish certain routines and precisely monitor ongoing progress and possible setbacks. What is more, the choice of Unity being a mature and well-known technology as a game engine proved crucial for speeding up the development, as the need to programmatically define graphics rendering or calculate collisions was delegated to the engine core and performed independently from the actual game logic.

Finally, it needs to be noted that the project described is by no means complete, as the implemented part was only intended to serve demonstration purposes. The game does not feature any narrative elements like dialogues or feedback on the player’s progress with the storyline. Certain bugs are present as well, especially the enemies shooting through walls may be seen at some point as the AI system was not fully optimized to function in narrow, twisted corridors. Moreover, game controls could be simplified by creating an interactive HUD with selectable fields instead of associating every action with a different key.

However, further expansion is possible as the game design and implementation allows for creation of new elements using the already complete mechanics, making it possible to grow the program into a full-blown arcade game. A number of prefabs representing walls, floors and other components was prepared, making it easy to build a new scene in a drag-and-drop manner. The same goes for all weapons, enemies and quest-related items like unlockable doors and hackable computers.

4.2. Appendix A – CD Content

The CD attached to this thesis contains materials in the folders structure

- **\DOC**
  - Electronic copy of the thesis
  - Game Design Document
  - Gameplay Design Document
- **\EXE**
  - Game in the executable format
  - **\Game_data**
    - All of the game's content including assets, resources, levels and compiled dll files
- **\SOURCE**
  - Full source code of "Laboratory Night"

4.3. Appendix B – Game User Manual

The player controls the main character using both mouse and keyboard. All the available controls ale listed below:

- **W** – moves the character forward
- **S** – moves the character backward
- **A** – moves the character to the left
- **D** – moves the character to the right
- **MOUSE** – moves the gunpoint
- **MOUSE – LB** – attacks with a basing weapon
- **MOUSE – RB** – attacks with the gravity gun
- **SCROLL** – changes between available weapons
- **1** – uses first skill, defensive shield
- **2** – uses second skill, black hole
- **3** – uses third skill, shock wave
- **SPACE** – activates devices, when available

The main goal of the game is to escape from the laboratory by beating enemies on all levels and reaching the final destination. It is possible to eliminate enemies using the basic gun, skills or the gravity gun. Third of the three possible choices is most effective. Certain doors are protected with passwords, the player needs to find and unlock appropriate computers.

5. Bibliography

1. [Chesebro89] Chesebro, J.W.,Bonsall, D.G., “Computer-mediated communication: human relationships in a computerized world”, University of Alabama Press 1989
2. [Loguidice14] Loguidice B., Barton M., “Vintage Game Consoles”, Focal Press 2014
3. [Ericson05] Ericson C., “Real-time Collision Detection”, Elsevier 2005
4. [Hart68] Hart, P. E.; Nilsson, N. J.; Raphael, B., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths"
5. [Reeves83] Reeves, W., "Particle Systems—A Technique for Modeling a Class of Fuzzy Objects," ACM Transactions on Graphics 2:2 (April 1983), p. 92
6. [Martin02] Martin R.C., “Agile Software Development: Principles, Patterns, and Practices”, Prentice Hall 2002
7. [Martin08] Martin, R. C., “Clean Code: A Handbook of Agile Software Craftsmanship“, Prentice Hall 2008
8. [Reeves87] Reeves, W. T., D. H. Salesin, P. L. Cook, "Rendering Antialiased Shadows with Depth Maps.", Computer Graphics (Proceedings of SIGGRAPH 87), 1987
9. [Crow77] Crow, Franklin C, "Shadow Algorithms for Computer Graphics", Computer Graphics (Proceedings of SIGGRAPH '77), 1977
10. [Ebert98] Ebert, Kenton Musgrave, Peachey, Perlin, Worley “Texturing and Modelling: a procedural approach”, Elsevier Science 2003
11. [Tobler06] Tobler, Maierhofer, “A Mesh Data Structure for Rendering and Subdivision”, 2006
12. [Akenine04] Akenine-Moller, Tomas, Haines, Eric “Real-time rendering”, A K Peters/CRC Press2004
13. [CodeConventions] C# Coding Conventions, https://msdn.microsoft.com/en-us/library/ff926074.aspx (*retrieved January 2016*)
14. [UnityScripting] Unity scripting reference, http://docs.Unity.com/ScriptReference (*retrieved January 2016*)
15. [NVidiaShadowMaps] Cascaded shadow maps, http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf (*retrieved January 2016*)
16. [Chan15] Smoothies, http://people.csail.mit.edu/ericchan/papers/smoothie/ (*retrieved January 2016*)
17. [CSGlobal] Hitboxes in Counter Strike: Global Offensive, http://blog.counter-strike.net/index.php/2012/03/1838/ (*retrieved January 2016*)
18. [BlizzardPress] Blizzard Press Center, http://blizzard.gamespress.com/WORLD-OF-

WARCRAFT-SURPASSES-10-MILLION-SUBSCRIBERS-AS-WARLORDS-OF-DRAE (*retrieved January 2016*)

19. [Somla] Unity infographic, http://blog.soom.la/2015/01/unity-infographic.html *(retrieved January 2016)*
20. [UnitySL] Unity docs, http://docs.Unity.com/Manual/SL-Reference.html *(retrieved January 2016)*
21. [UnityGraphics] Unity docs, https://Unity.com/learn/tutorials/modules/beginner/graphics/lighting-and-rendering *(retrieved January 2016)*
22. [UnityLighting] Unity docs, http://docs.Unity.com/Manual/Lighting.html *(retrieved January 2016)*
23. [Mathis15] Low Poly Character Modelling by Ben Mathis, http://www.3dtotal.com/team/Tutorials/benmathis/benmathis_1.asp (*retrieved January 2016)*
24. [UnityMaterials] Unity Manual – Material Parameters, http://docs.unity3d.com/Manual/StandardShaderMaterialParameterNormalMap.html *(retrieved January 2016)*

6. Index of figures

Illustration 1. Screenshot from the game Space Invaders. 8
Illustration 2. Market share of game engines. 11
Illustration 3. Unity Editor. 13
Illustration 4. Structure of meshes. 14
Illustration 5. Model with small number of polygons. 16
Illustration 6. Simplified hand model from open source project Sintel. 16
Illustration 7. Blender Editor. 17
Illustration 8. Bump mapping. 19
Illustration 9. Object without (left) and with (right) normal map applied. 20
Illustration 10. Object before (left) and after (right) performing parallax mapping. 21
Illustration 11. Sample mipmap. 22
Illustration 12. Sample bitmap used in anisotropic filtering. 23
Illustration 13. Different lighting effects. 24
Illustration 14. Diffuse reflection. 25
Illustration 15. Lambertian Light. 26
Illustration 16. Specular highlight. 26
Illustration 17. Screenshot from the game SOMA. 27
Illustration 18. Process of Rasterization. 31
Illustration 19. Path taken by light. 32
Illustration 20. Left- and right-hand coordinate systems. 33
Illustration 21. Sample state diagram for an enemy character. 42
Illustration 22. Commit history in a Git version control system. 45
Illustration 23. Sample gun mesh from “Laboratory Night”. 49
Illustration 24. A raw model (left), with applied texture (middle) and normal map (right). 50
Illustration 25. Light object properties. 51
Illustration 26. Point light applied in “Laboratory Night”. 51
Illustration 27. Spot light applied in “Laboratory Night”. 52
Illustration 28. Directional light in “Laboratory Night”. 52
Illustration 29. Area light model. 53
Illustration 30. Selection of a rendering technique. 54
Illustration 31. In-game objects equipped with box (left), sphere (center) and capsule (right) colliders. 56
Illustration 32. Navigable area in a pathfinding system. 57