jamk.fi

Large boss characters in Unity engine

Reimplementation of gameplay mechanics from Shadow of the Colossus video game

Matej Isteník

Bachelor's thesis
May 2015

Degree Programme in Software Engineering
Technology, Communication and Transport

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Description

Author(s)	Type of publication	Date
Isteník, Matej	Bachelor's thesis	25.05.2015
		Language of publication: English
	Number of pages: 36	Permission for web publication: x

Title of publication

Large boss characters in Unity engine

Degree programme

Software Engineering

Tutor(s)

Salmikangas, Esa

Assigned by

Zaibatsu Interactive Inc.

Abstract

The goal of this thesis was to implement the gameplay mechanics from the Shadow of the Colossus video game in the Unity engine. First released for PlayStation 2 in 2005, this game is centered around the player character, human male named Wander, who battles large being known as the colossi.

The main technological achievement of the Shadow of the Colossus and the difference compared to other games is the design of the colossi. In the game, the player character is allowed to move on or to climb the body of the colossi, as their collision shapes are fully modeled and animated. As the collision detection and resolution can be expensive to compute, they are usually simplified in most of the games by using a primitive shapes or static geometry, which is also the case with the Unity engine, which does not support dynamically changing collision meshes.

For these reasons, alternative implementation of needed mechanics were implemented, such as the custom dynamic mesh collider component, its skinning and animation component and the player character component. They were designed to use custom collision detection and resolution mechanics, however, otherwise the use the available components and technologies provided by Unity, such as animation and bones support or rendering of the skinned meshes.

Keywords

boss, large, collision, animation, bones, mesh, skinning, Shadow of the Colossus, Unity engine

Miscellaneous

Contents

Figures

Glossary

Bone

In 3D graphics, part of the virtual skeleton used to deform the shape of the object.

Boss

Special game character used in boss battles, often with unique shape, behavior or abilities.

Bounding sphere

Bounding volume in the shape of a sphere.

Bounding volume

Volume, which contains the bounded object.

Collider

Shape which represents the object in collision calculations.

Collision engine

In simulation, system used for detection and resolution of collisions between objects.

Colossus

Colossal creature from the Shadow of the Colossus video game.

Frame

Point in time, when the game state is updated.

Game engine

Software framework used by video game, which contains various subsystems, like graphics rendering, physics calculation, input handling or networking.

Mesh

Data structure, which describes 3D object with vertices, edges, and triangles. Most commonly used representation of 3D objects in video games.

Platforming game

Video game focused to movement mechanics, such as running, jumping or climbing, where the main task of the player character is to overcome various terrain obstacles.

Player character

Character in game, which is controlled by the player.

Polygon

Closed shape defined by multiple points.

Skinning

In 3D graphics, process, where the mesh shape is deformed in relation to the movement of its bones.

Triangle

Polygon formed by three points.

Unity

Modern and advanced game engine commonly used today.

Vertex

Point in three or two dimensional space, which is used to define polygons.

1 Introduction

One of the typical gameplay mechanics in the video games are boss battles. The boss battle is usually a special event where the player character battles against an uncommon enemy. This can be either much stronger enemy with more health and attack power than common enemies or the fight can use special rules and be divided to several phases.

In some games the boss battles are designed as a puzzle. Boss can attack or behave in some predefined pattern and reacts to player actions. Player is expected to study how this mechanics works and to figure from the observation how to defeat the boss. Some requirements have to be respected; for example, the behavior pattern of the boss have to be observable and predictable (to some degree). In bad designs, the gameplay can become too chaotic or random, negating the requirement of the observation of the boss. The observation phase can be simplified by directly provided hints to the player, as is more common in newer games, or indirectly by clues which can be found during the game walk through.

As the game bosses usually differs from the common enemies, one of the recurring characteristic of them is the size difference in comparison to the player character, where the boss can be much larger. Another characteristic can be the amount of its raw power, either in damage output of his weapons or the amount of health.

Figure 1: Final boss battle in Jamestown.

This is also often the case when the boss fight is designed as a puzzle, so the frontal assault against the boss is not possible or feasible. For example, in the video game Jamestown, the fight against the final boss is divided to two parts. In the first part, the boss alters between a phase when he is vulnerable and a phase when he covers itself in an impenetrable armor and launches an attack against the player character. After he is defeated, he changes the form and second phase began. This phase can be seen in

the figure 1 (page 1). The boss is located in the center and location of the player character is in right bottom. In current situation, the boss is invincible. The player have to attack the cubes which floats around the boss. When the cube is damaged, it will collide with the boss and damages him. When this happens the boss retaliates back with a strong attack against the player. In this case, the cubes can be instead used by the player as a shield. The battle ends when the boss health (yellow bar near the top) is depleted or when the player character dies.

As bosses in boss battles can be quite large and their behavior much more complex in comparison to common enemies, they also brings some technical challenges. For example, Nintendo Entertainment System game console allows only limited amount of sprites to be shown at the same time and their maximum size was constrained. To overcome this limitations, the boss was usually drawn using background tiles. Because of this, the actual background of the battle scene was just black color. This can be seen in the figure 2 (page 2), where the boss (green dragon) is quite large in comparison to the player character, the background is black and the blocks on which the player characters stands are actually created using a sprites (D'ANGELO, 2014).

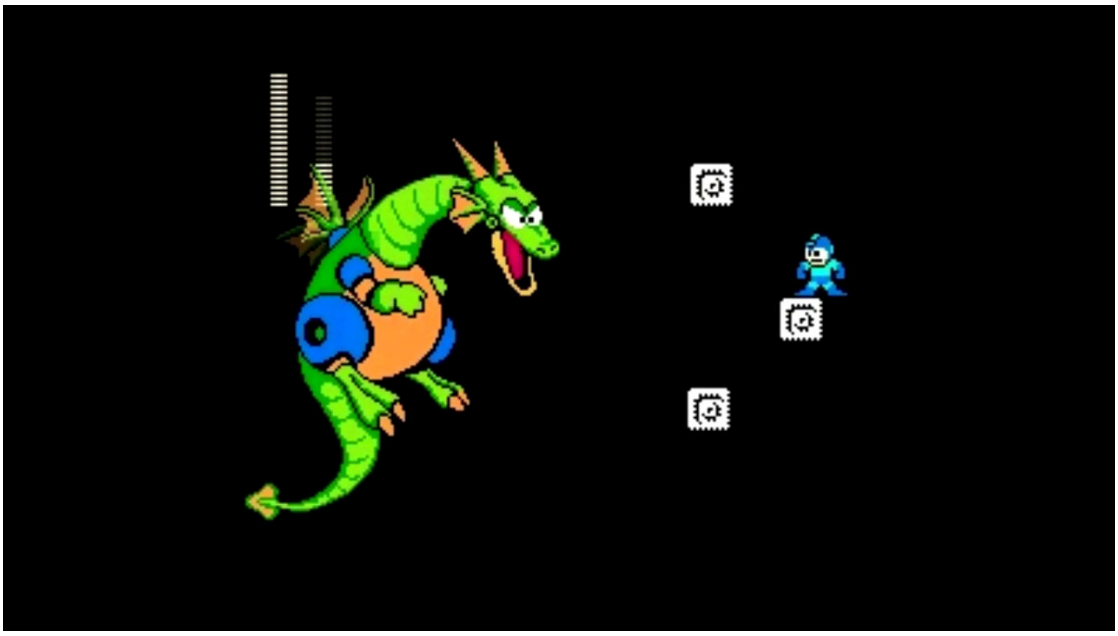

Figure 2: Boss battle in Mega Man 2.

The goal of this thesis is to implement the boss battle mechanics from the video game Shadow of the Colossus. The boss battles in this game were puzzle based and their implementation marks it as one of the most advanced video games on its original target platform. The thesis will focus on implementation of the technical aspects of Shadow of the Colossus game in the Unity engine. This work is done for the Zaibatsu Interactive Inc.

2 Shadow of the Colossus

2.1 Introduction

Shadow of the Colossus is an action adventure game, released for PlayStation 2 console in 2005. The gameplay is centered around the fights against a large beings, known as colossi, with the player character, a human named Wander. Every fight against a colossus can be classified as a boss battle, with the observation mechanics and boss fight phases.

Figure 3: Player character and a colossus.

2.2 Gameplay mechanics

Colossi are huge creatures of various forms, usually shaped as humanoids or animals. Their sizes are between 30 to 300 meters in height or length. They are composed of flesh, usually covered with fur, and stone parts, formed in shape of an armor or weapon. Typical look of a colossus and its comparison against the player character can be seen on the figure 3 (page 3). Every colossus has some number of weak points. These points can be stabbed by sword, thus lowering the amount of colossus hit points. Each weak point can absorb only a limited amount of damage. When this amount is used, the weak point will disappear and is either replaced by another in other location, or the colossus will die, if it was the last one. The player can locate an active weak point by using a mini game: when raising his sword a number of light rays will show, concentrating on the direction to the next point. Because of this, some time is usually spent exploring the body of colossus for finding the active weak point.

For approaching these weak points, a player character has to literally climb the colossus, which is due to its massive size. As every colossus is different in shape or behavior, and the battle occurs in different places, the way to climb a colossus changes as well. Moving on the colossus follows the same principles as moving on terrain, and is similar to mechanics used in other platforming games. The player can move on horizontal surfaces, climb over ledges, formed by a colossus's stone parts, or grab on and climb a vertical surfaces covered with colossus's fur or vegetation. One of the unique feature of the game is in the combination of these mechanics with colossus's dynamic, moving body.

A player can freely navigate on colossus restricted just by its shape and availability of fur. As the colossus is moving, passable surfaces can become impassable. The colossus also actively tries to shake the player down. This can be prevented by either standing on appropriate surface, or tightly holding on its fur. The player is limited by the amount of energy used for continuous holding. If this energy is depleted, the player character will release its grip and usually falls down from the colossus. Energy can be replenished by staying on safe (usually horizontal) surfaces (ADAM SIDDIQUI, 2011, .)

2.3 Target platform

During the time of its release, Shadow of the Colossus was exclusive PlayStation 2 game. It was released in the end of the life cycle of this console, therefore the developers were quite familiar with its architecture and were able to push it to its limits. The console was based on custom components, as a special 300 MHz processor (named Emotion Engine), 32 MB RAM, 4 MB of fast eRAM and graphical unit (named Graphics Synthesizer) loosely comparable to GeForce 3 or ATI R100. Even with these limitations, developers working on Shadow of the Colossus were able to produce a game almost comparable in terms of graphics and physics simulation with the first games for the 7th generation of game consoles. This achievement was not without sacrifices, and the frame rate of the game could drop to as low as 15 fps. On the other side, common hardware today, even in mobile devices, is often much more powerful, so techniques used for this game can be easily applied in larger scale without performance problems (UEDA, SUGIYAMA, SEKI & TANAKA, 2005).

2.4 Colossi

Colossi in Shadow of the Colossus have various shapes, forms and sizes, but they are animated in same way, using the skeletal animation and deformable meshes. Body of every colossus consists of:

- bones
- collision mesh
- visual mesh

Bones form animation skeleton of a colossus. The shape of colossus's body changes in relation to their position or orientation. Bones affects both visual and collision mesh. The collision mesh is a simplified mesh, which represents the colossus's shape in physics and collision calculations. Visual mesh is much

more detailed mesh, with additional information, e.g. textures or UV coordinates, used for graphical representation of colossus. Shapes of both collision and visual mesh are deformed by bones in similar way, so they can stay in sync.

Figure 4: Player character in an unstable state.

2.5 Player character

Player character is a human male of usual size. In terms of gameplay he can be controlled to perform various typical actions for platforming games such as running, jumping or climbing. Player character can be in various states where every state allows different subsets of available actions or level of control of the character. In a normal state, the player has full control of the character. He can control it to move towards a specified direction, jump, climb or use weapons. In this state, collision and movement are handled similarly to other games and the player character collision shape is modeled as a sphere shape. Another states occurs when a controlled movement of character is not possible. This can occur for multiple reasons; the player character falling, tumbling or hanging. Many times these states occurs because of a colossus's movements. As a colossus is a dynamic object, some surfaces can became impassable, or its movement can be too fast and erratic. When the player is holding on to colossus by its fur, but cannot be supported by its feet, his body is modeled as a pendulum. This can be clearly seen on the figure 4 (page 5), where the player character hangs on the neck of the colossus. One part of the pendulum is fixed and is located near the character's palms, with the other part, dynamic, ends near his feet. In this state, the character is animated using inverse kinematics to follow the pendulum movement.

2.6 Colossus and player character interaction

Collision in computer games usually means a simulation of physical interaction between multiple objects when the volumes of the objects are going to overlap. There are various approaches how to handle this situation, depending on available technologies, computational power and requirements. Because of the nature of video games, algorithms for this task have to run in a very limited time frame and be as fast as possible, often sacrificing precision.

Most of the games calculate collision using primitive shapes for dynamic objects (characters, projectiles) and more complex and detailed shapes for static objects (terrain, buildings). This is based on common optimizations of collision handling. Primitive shapes have fast and mathematical ways to detect collisions. Static structures can be partitioned using proper data structure, thus it is possible to quickly filter parts against which the collision can occur, and run the detailed collision test only against them. This also brings constraints, as not every object can be abstracted by primitive shapes, and commonly used data structures for space partitioning are usually optimized for static geometry.

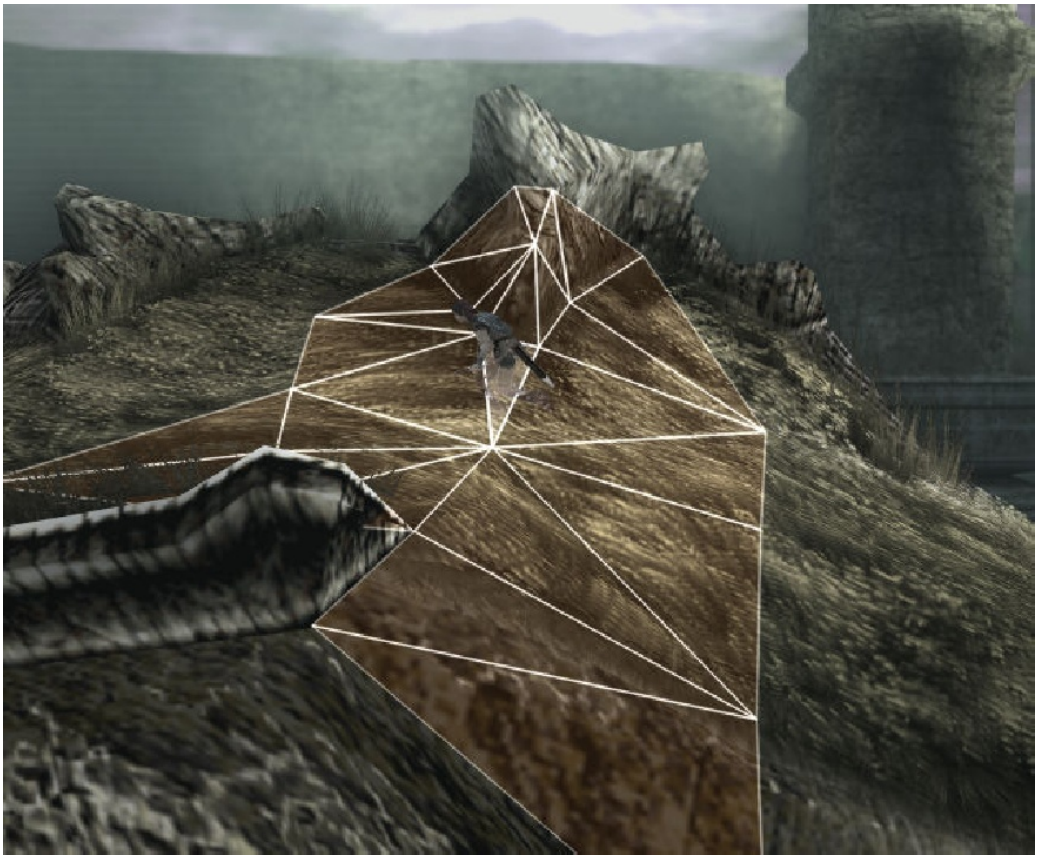

Figure 5: Movement of the player character on a colossus body.

As can be seen on the figure 5 (page 6), in Shadow of the Colossus every colossus is technically a dynamic, changing terrain, on which the player character can move. This brings much more constraints to the used data structure, as not only retrieval of objects but also modifications of their positions have to be fast. As the shape of a colossus can change in an organic way (its surfaces can stretch or bend like a skin), player character interaction with the colossus is more complex than in most other games.

2.7 Similar games

2.7.1 Comparison

Few games used various gameplay mechanics from Shadow of the Colossus or are similar in technical aspects, but so far, no game can be considered as a real successor. Usually the used gameplay mechanics are much more simplified in comparison to Shadow of the Colossus.

2.7.2 Castlevania: Lords of Shadow

This game contains bosses, named titans, with similar gameplay mechanics to the colossi from Shadow of the Colossus. The goal of the player character, normal sized human, is to approach titan's weak points and destroy them in order to defeat it. This can be seen on the figure 6 (page 7), where the player character is going to smash one of these points. Movement mechanics are much more simple than in Shadow of the Colossus. The movement on the titan's body consists of holding on ledges and jumping between them. The path which the player can take is limited, much more linear, often combined with quick time events for advancing to another part of the titan (for example, jumping on titan's hand when it is near the player). Player is also guided by game hints, so the exploration phase is much simpler than in Shadow of the Colossus (OMEGAEVOLUTION, 2010, .)

Figure 6: Titan boss and the player character in Castlevania: Lords of Shadow.

2.7.3 God of War III

Some of the bosses in this game are large, hundreds of meters high beings from Greek mythology, Titans. In one part of the game, player character is faces titan Cronos. The player size is again the size of an ordinary human, as can be seen on the figure 7 (page 8), where the player character is located on the

bottom left part of the figure. Fight with this titan is based on moving on his body through a number of phases which are scripted and linear in nature. The player is always restricted to some part of Cronos body, however he or she can freely move on it. Gameplay consists of switching between fighting with various lesser creatures, popping form the titan's body, and climbing parts, where the player advance through physical obstacles. This is combined with quick time events, for example throwing hooks to various points for swinging the player character to another place (OLYMPIANDAWN, 2012, .)

Figure 7: Cronos boss and the player character in God of War III.

3 Unity engine

3.1 Introduction

Unity is a modern game engine developed by Unity Technologies. It is one of the today most used game engines, often selected by small or medium development teams. It provides usual components for the video game, like 2D/3D rendering, collision, physics and input handling and others. It also provides simple and powerful scripting interface to access these components. Available languages for scripting are C#, UnityScript and Boo. Unity supports a large variety of target platforms, from desktop and consoles to mobile devices.

3.2 Physics and collision detection

For physics simulation of 3D objects, Unity provides wrapper around PhysX library, which is today one of the most advanced physics simulation engine. Physics engine in Unity is used for these main tasks:

- Physics based motion of objects.
- Collision detection and resolution.
- Fast search and query for the objects in relation to their position and shape.

In Unity, the object which actively participate in physics simulation has to contains Rigidbody component and one of the available collider shapes component. Rigidbody can be set to two modes: kinematic and non-kinematic. Non-kinematic rigidbody behave similarly as an object in real world, where its position or direction is affected by physical forces. It interacts with other rigidbodies and can collide with them. Kinematic rigidbody affects other objects, but its position or direction is not affected by other objects. This mode is useful for animations or scripted movement. The last case is an object which contains a collider, but not the Rigidbody component. This is used for static objects, often terrain or environment, so non-kinematic rigidbodies can collide with them.

The shapes of objects used in physics simulation are often a simplified representation of visual shape. There are two main reasons:

- Less demanding for calculations.
- Sufficient for most situations.

For example, in the case that bodies of a humanoid characters should not collide with each other, just a simplified shape as circle, box or capsule can be used, as can be seen on the figure 8 (page 10). In other cases, much more detailed collision information can be needed, such as detection of which part of an object participate in the collision (humanoid's arm, vehicle's component, etc). In this case, mesh colliders or multiple primitive colliders (known as compound collider) can be used to form the shape of the object, as can be seen on the figure 9 (page 10).

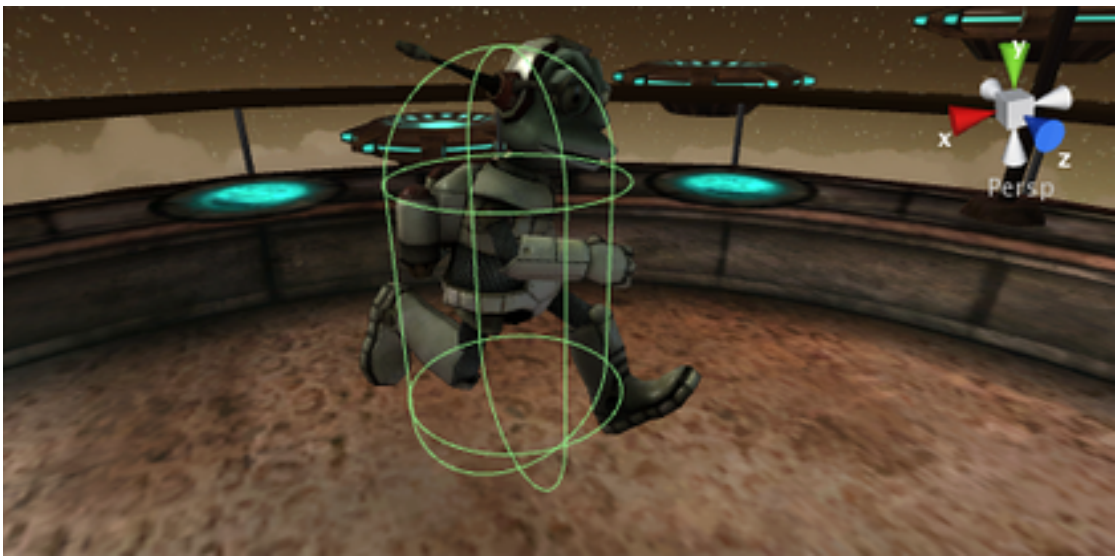

Figure 8: Character model and its simplified collider.

Unity provides primitive colliders the most common shapes and also the mesh collider. Primitive shapes can be dynamically changed or moved and the physics simulation will behave correctly. This also applies to mesh colliders which are set as convex. Because of some certain properties, convex objects are more suitable for collision detection algorithms. For example, to detect if a point is inside a convex mesh or if two convex meshes collide. If a mesh collider is set as convex, but its mesh shape is concave, Unity will create convex mesh bounds which contains and to some degree follows the shape of the original mesh. Mesh collider set as non-convex cannot be moved, as the underlying space partitioning and search data structure is not designed for fast movement of complex, large objects. Shape of convex Mesh collider cannot be also directly changed, for the same reasons (UNITY PHYSICS) (UNITY MESH) (UNITY COLLIDER) .

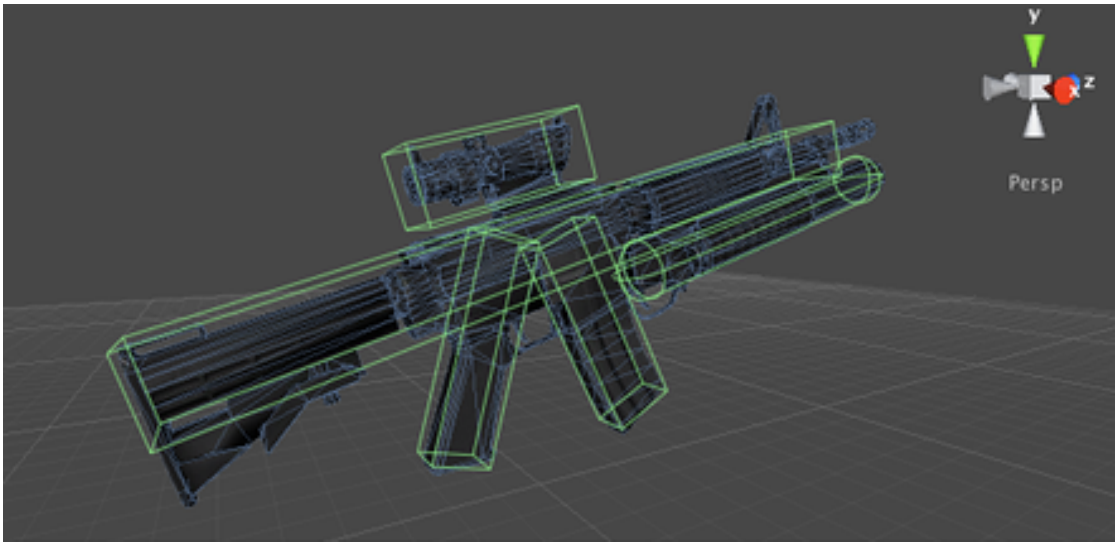

Figure 9: Compound collider from multiple primitive shapes.

Figure 10: Object rendered with the Unity Skinned Mesh Renderer.

3.3 Mesh animation

As colossus object is composed of parts, which can stretch and bend, appropriate rendering mechanism which supports the deformation of the mesh is needed. Unity provides Skinned Mesh Renderer component, which uses a combination of bones and mesh's vertex groups. The mesh is deformed in relation to the position and orientation of bones. Animations can be made by manipulation of these bones, using either forward or inverse kinematics. Forward kinematics is based on setting the position and rotation of the bones directly to create the desired pose. On other side in inverse kinematics, only final position of the limb tip is defined, and the solver will try to position and rotate the bones of the limb. In Unity, the inverse kinematics is supported only for humanoid skeletons. Example of the mesh rendered by Skinned Mesh Renderer can be seen on the figure 10 (page 11) (UNITY SKINNED MESH RENDERER) .

3.4 Player character control

Usually two ways how to control the player character are used in Unity:

- Character Controller component.
- Rigidbody component with associated Collider component.

Character Controller component is built-in component for character movement, which respect the collision with other objects. A character controlled this way can interact with physics objects (for example, pushing them), but is not affected by them. This is obvious when the character of a player tries to stand on a moving platform. In this case, the platform will move, but character's body will keep its position.

Alternatively, Rigidbody component can be used, either set to non-kinematic, using forces for its movement, or kinematic, where the position and rotation are set directly. Rigidbody brings better mechanics for interaction between the player character's body and other rigidbodies, but on other side, non-kinematic movement can be harder to implement properly.

4 Design

4.1 Introduction

Unity provides components for player character control, physics, collision or animation. Unfortunately, using them for recreation of the game mechanics of Shadow of the Colossus, centered around the interaction between the player character and a colossus, is problematic, as they have various constraints and limitations.

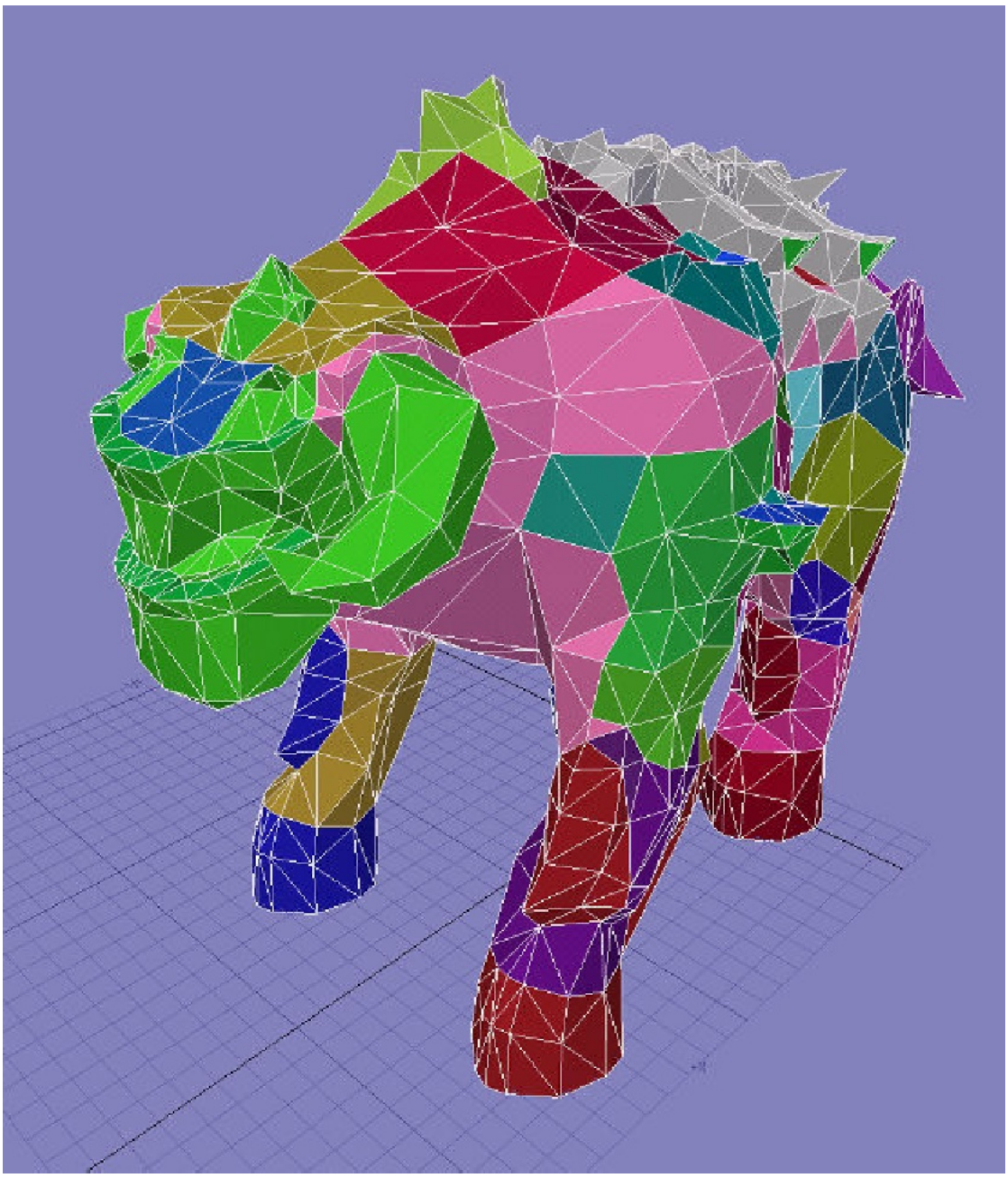

Figure 11: Collision mesh of a colossus.

4.2 Unity collision system

In simulations, only necessary things are simulated or modeled. The same applies to video games, especially when collisions are involved. In Shadow of the Colossus, the main aspect which dictate how the collisions are to be calculated, is the size proportion between the player character and colossi. As every colossus is much larger than the player character, and the player character should be able to move on it, its collision shape should be similarly detailed as another passable terrain, what can be seen on the figure 11 (page 13). Because colossus is also a dynamic object, it should be possible to change its shape to various poses, visible on the figure 12 (page 15). This means that proper interaction between a colossus and the player character should be modeled. Implementation should cover cases as when the shape of the colossus changes and it starts colliding with the player character or the proper positioning of the player character when colossus's skin stretches or bends. The player character can use simplified shape, as capsule, because it should be suitable for this task.

Because of previous needs, these requirements are needed from collision system of a colossus:

- Shape represented by a mesh based collider.
- Possibility of fast change of the shape.
- Proper interaction between the collider and other colliding objects when the shape of the collider changes.

Unfortunately, built-in collision and physics system in Unity is not suitable for this needs. The reasons for this are:

1. Unity Mesh Collider has to be convex, otherwise it cannot be moved.
2. Change of the shape of Unity's Mesh Collider is not properly documented and can produce unexpected results or costly recalculations.
3. Unity physics API does not support deformed or stretched surfaces.
4. Actual deformed shape of the mesh produced by Skinned Mesh Renderer is not easily accessible.
5. The update of the mesh date cannot be as fast as possible, because whole data (vertices or triangles) are always copied to and from the mesh.

Points 1 and 2 rules out usage of built in Unity Mesh Collider as the collider for colossi. Concave mesh colliders in Unity are optimized for static geometry, such as terrain. Even if the colossus object is built with multiple convex colliders, dynamic change of their shape can be problematic. Official documentation does not cover the case of modification of the collision mesh. With some experimentation, this is possible, but without knowledge if and how the change is processed inside physics subsystem.

Point 3 is closely related to points 1 and 2. Change of a collider shape, either scaling or manipulation with vertices, is not reflected in the physics engine. This functionality is needed to deform the shape of the colossus, so the player can properly interact with its shape. Unfortunately, these three points rules out the possibility of using the built-in physics to simulate the movement of the player character.

Point 4 brings on another complication. During animations, the collision mesh should follow the shape of visual mesh. Unity supports bone animation, however it is not possible to quickly access current positions of vertices. Skinned Mesh Renderer is calculates the deformation of mesh on CPU, but results are sent directly to GPU. It is possible to bake a current mesh state, but as it involves retrieval of the data from GPU memory, it is very slow. This rules out the usage of Skinned Mesh Renderer for deformation of collision mesh. This means that an independent algorithm for mesh deformation needs to be used in conjunction with Skinned Mesh Renderer component. It also has to follow the same or very similar mesh deformation rules, otherwise the visual and the collision mesh will not be synchronized. As deformation is based on bone weights and vertex groups, it should not be a problem to implement it and follow the behavior of the Skinned Mesh Renderer.

Point 5 should not be a performance problem, but because of the previous reasons and simplicity of storage of the mesh data (arrays of vertices and triangles), more suitable implementation for their storage can be used. Built-in Mesh component can be still used to define the collision mesh shape.

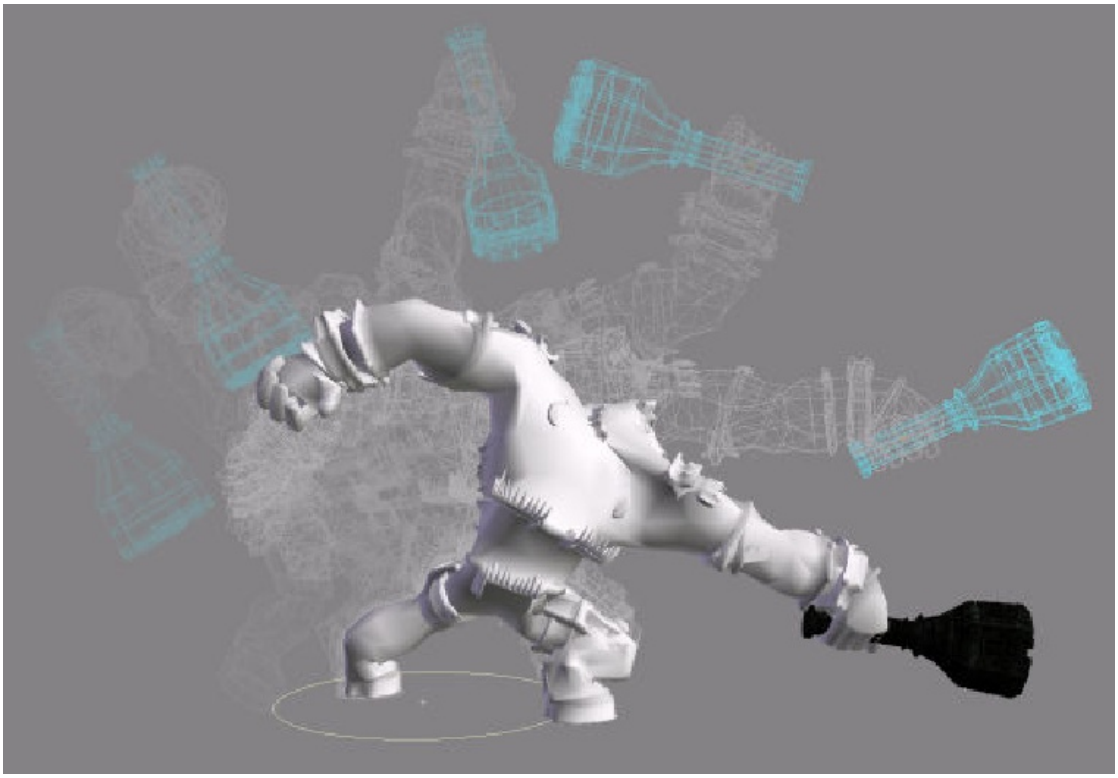

Figure 12: Shape change of a colossus during animation.

4.3 Generic collision mechanics

4.3.1 Introduction

In a trivial collision algorithm, collision tests are run between each object in the scene. As an object can be constructed from multiple colliders, the number of tests can be $m \cdot l$. In the worst case, this can lead to algorithm complexity of $O(n2 \cdot m \cdot l)$. If proper techniques are used, it is possible to run the collision tests much faster. In optimized collision test algorithms, two principles are used:

- Minimization of the amount of collision tests against the other objects.
- Usage of suitable collision shapes and highly optimized algorithms for the collision tests.

Both principles deal with the fact, that it is easier to find if a collision cannot occur, than if it occurs and what are their parameters. If proper data structure is used, it is possible to partition Cartesian space in a way, that it can be quickly detected if some group of objects cannot collide with another group. Also, if complex shapes are bounded by primitive ones, it is possible to first test the collision between the primitive bounds, and run detailed collision test only afterwards.

4.3.2 Position based search data structures

Data structures used for space partitioning are able to query objects in relation to their position and bounds. For example, retrieval of the closest object to a specific position or all objects which are located in specified volume of space. This way, it is possible to filter the objects against which the collision test is run. Search trees are often suitable for this task, where typical example is the Binary Space Partitioning (BSP) structure. Other variants are Octree (example shown on the figure 13 (page 16)), kd-tree, R-tree, AABB, etc. They are different in a way how the shapes can be retrieved or in their orientation towards dynamic or static geometry.

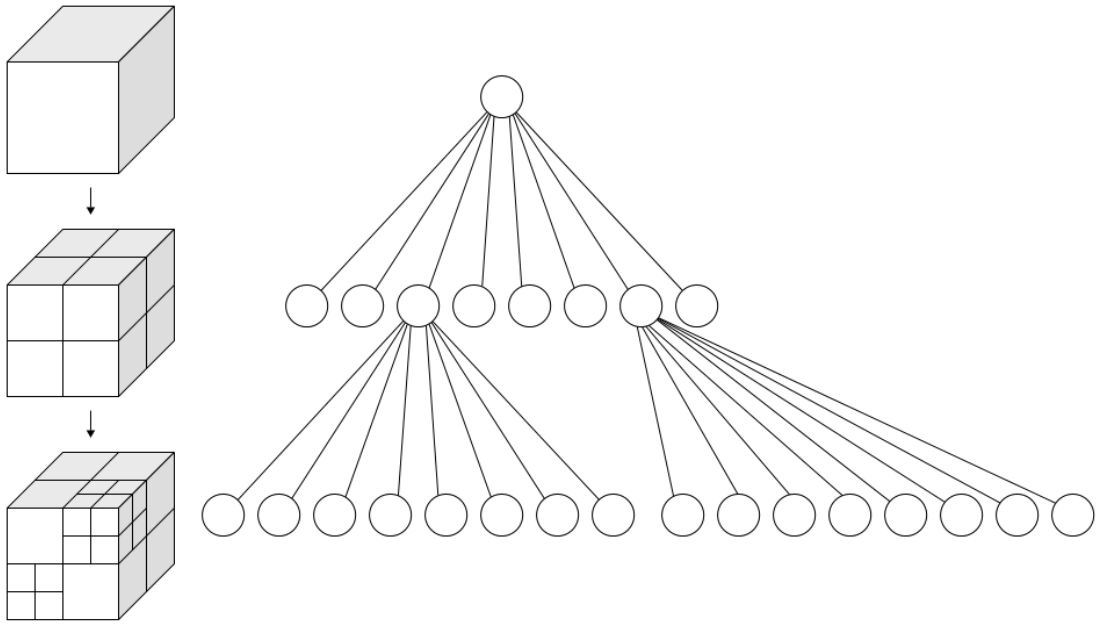

Figure 13: Principle of Octree space partitioning data structure

4.3.3 Collision between the collider shapes

A trivial way how to represent the collision shape of an object is to use a mesh collider, as it can closely follow the real shape of an object; however in many situations, this is not necessary and for significant speed up only primitive collider can be used.

Typical primitive collision shapes are sphere, capsule and box, with their 2D equivalents, circle and square. Their advantage is, for testing if the collides, mathematical equations can be used. In case of spheres, only radii and distances are important, and collision can be determined with simple mathematical operations. Capsules works in similar way. Boxes are usually axis-aligned (sides are perpendicular to Cartesian axes), so the collision test can be simplified. Tests between the different shapes are more complex, as no universal solution or equation exists, and almost every case have to use different algorithm.

Collision between mesh colliders is much more expensive. There are two cases: convex and concave colliders. In case of convex mesh shapes, it is possible to use their projections on a plane. If a alignment of projections exists, where they does not overlap, the objects does not collide. In case of a concave collider, all triangles have to be tested against each other, what can be quite slow for some applications.

One way how to speed up collision tests is to bound mesh colliders inside primitive colliders. This way, tests between primitives can be carried first, and only if a collision between them occurs, more detailed collision test is run between the colliders.

Colossus can be modeled either with a concave mesh collider, or with multiple convex mesh colliders. In this case, they have to stay convex even during animations. As it can be problematic to keep this constraint and because with good algorithms it is possible to quickly discard triangles which can not collide together, concave mesh collider is preferred to represent the shape of a colossus.

4.3.4 Bounding volumes partitioning

As the body of a colossus is dynamic, it can problematic to use space partitioning data structures usually designed for static geometry. A simple way how to overcome this obstacle is to wrap the colossus's bones, which represents the body parts as torso, head, limbs and other, inside a primitive bounding volume, such as sphere. Position of these bounding volumes will follow the movement of associated bones and they can be placed in appropriate data structure for fast search or retrieval. Collision test between an object and a colossus can be first run against these bounding volumes. If a collision can occur, the test will be run against the triangles, which forms the part associated with the bounding volume.

4.3.5 Movement of objects and collision resolution

Algorithms for collision detection are divided to two categories:

- discrete
- continuous

They differ in a mechanism, how the collision is detected. In discrete collision detection, the collision detection algorithm between an object and an obstacle is run after the object is moved. Because of this, the object shape can overlap with the obstacle shape before the collision is resolved. On other side, continuous detection detects the collision in advance, and moves the object only to the closest position,

where it touches, but do not collide with the obstacle. This type of detection is usually more algorithmic expensive to detect collision, but it allows more precise collision resolution.

Important disadvantage of the discrete collision detection is the bullet-through-paper problem. In some cases, when the object moves too fast, it can pass through other objects. This can happen because the collision detection is run after the movement. When the object is moved, it is directly placed to the new position computed in relation to its initial position, velocity and the time difference between this and the previous frame. If there is too thin or small obstacle between the old and new position of the moved object, the collision between it and the obstacle can be missed, as the moved objects does not collide with the obstacle object in its new position. This is shown on the figure 14 (page 18), where the sphere was moved from the position S to position E without collision with the obstacle.

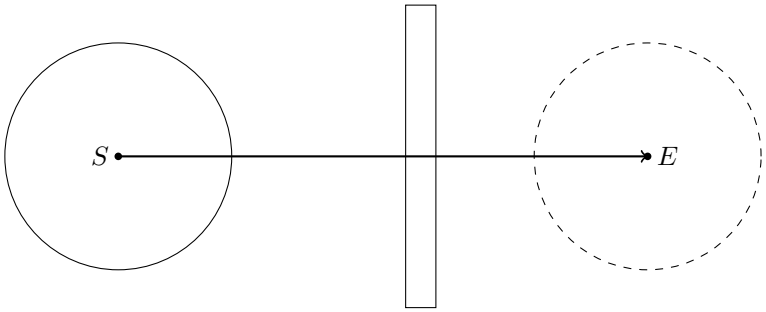

Figure 14: Bullet-through-paper problem example.

There are various ways how to solve this problem. Trivial approach is to run multiple collision tests as the object is incrementally moved towards its new position. The distance between the tests is related to the size of the object, so the next collision test partly overlaps with the previous one.

Another approach is to use the continuous collision detection and the technique called speculative contacts. In this approach, ray casting is used to determine the distance between the moved object and the approximate contact point on an obstacle. If the contact is detected, position on the ray where these two objects touches is found and the object is positioned on it, as can be seen on the figure 15 (page 18). There, the collision position of the sphere, which travels from point S to point E, and the obstacle was calculated as the point $E1$. This approach works well for simple obstacle objects, such as boxes, but it can be hard to implement in collision with more complex geometry, as it can be quite complicated to find proper position of point $E1$ (FIRTH, 2011).

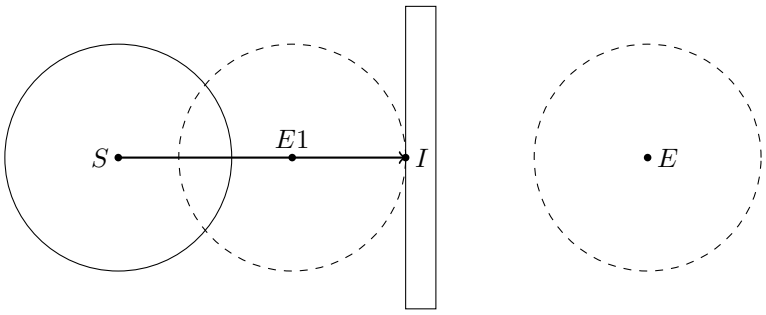

Figure 15: Continuous collision detection.

There are more problems with collision resolution. One of them is the motion bleeding. In simplified system, when the moved object collides with an obstacle, it is not bounced back, but only placed on a position where it touches the obstacle. In a more complex system, in relation to the surface materials and the weights of the objects, the bounce force is calculated and applied.

Another problem is the possibility of introducing new collision when current collision is resolved. It can happen, that after the object is moved to a new position, it will collide with another obstacle. There are multiple approaches how to deal with this problem. Trivial solution is to do nothing; the new collision will be resolved in the next frame. Another trivial solution is to run another collision test after the collision resolution, and repeat this step until no new collisions are detected. Because it is possible that objects is stuck in a situation, when it is not possible to resolve the collision without introduction of a new collision one, only limited number of tries is run in one frame. This case can be seen in the figure 16 (page 19), where the circle with the center S is stuck between two obstacles and it is not possible to resolve collision with one obstacle without collision with another one.

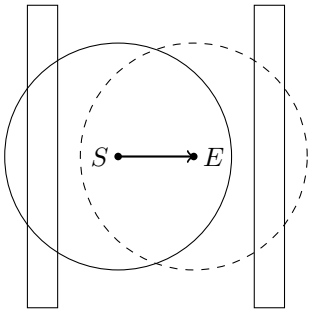

Figure 16: Unsolvable collision.

There is also a causality problem. During collision resolution, object can move in that way, that it will affect the motion of other object, which should not be affected. For example, its new position can be in direction of the motion of another object, but in a time when the other object should already pass that position. Because collision resolution for the second object is run after the first, its movement will be affected by collision with the first object. This problem can be solved with calculation of all the collision resolutions first, and move the objects afterwards.

Many of these problems can be easily eliminated or are important only in specific environment. In our case, the environment consists of the colossus, the terrain and the player character. As the speed of the player character and colossus movement are similar, chance of bullet-through-paper is minimal and can be solved by adjusting the physical time step value. The weight difference between the colossus and the player character is significant, so when collision occurs, it is safe to treat colossus as static object. In typical case, only 1 colossus and 1 player character will interact on one scene, so the causality problem also should not be a problem.

4.4 Animation of the mesh collider

As it is not possible to use Unity built-in Skinned Mesh Renderer to animate the mesh collider which represents the colossus shape, and Unity does not have any similar equivalent, it have to implemented independently. This mechanism should follow deformation mechanics used by Skinned Mesh Renderer, so both visual and collision meshes will be synchronized.The deformation is based on bones, vertex groups and weights. Every vertex can be influenced by multiple bones. Each vertex has defined weight value for each bone, by which it can be influenced. The sum of this values is always 1. Vertex is then positioned in relation to bones position, orientation and weight values.

For requirement of the physics simulation, additional parameters concerning the triangles have to be calculated: velocity and surface shape change. Velocity can be used in more accurate collision detection and resolution between the object and the colossus. Shape surface change is used in calculation, if an object in contact with the colossus should be translated, when shape of colliding triangle is changed. Example of this is the player character which stands on the colossus, and the skin of the colossus stretches or shrinks during animation.

4.5 Terrain and player movement

Figure 17: Wall climbing in the Shadow of the Colossus.

Player character movement should be similar to that in other games. It should run, jump, climb over the ledges or on suitable surfaces, as is shown on the figure 17 (page 20), where the player character from the Shadow of the Colossus climbs the wall. Differences are, that some of the surfaces or ledges can change their orientation, and thus become inaccessible or impossible to hold on. When standing, player should have defined maximum slope of surface on which it can still stand. If this angle is too big, a character will start sliding down. In case of a ledge, there are two options: either the ledge will become too steep to hold on and player character will fall down, or, if rotated in opposite direction, ledge will become walkable surface. In that case, the player character will switch from holding on ledge to standing. Another case is, when the shape of an object is deformed, for example stretched. In that case, the player character will be moved proportionally to how the surface shape changed.

5 Implementation

5.1 Main goal

Thesis focus on modeling the colossi and their interaction with the player character. This consists of implementation of the collision and animation system which works with dynamic mesh collider and proper movement mechanics and interaction between the player character and the colossi or other objects.

5.2 Dynamic mesh collider

5.2.1 Introduction

The Dynamic Mesh Collider is a custom component which aims to provide collision mechanics between a primitive shape and a dynamic collision mesh. As it is currently not possible to extend or modify the built-in Unity collision mechanics, the implementation of this component is independent from them. It supports collision detection against a sphere shape and ray casting against its shape. This functionality provides enough data for a collision evaluation and resolution. It allows dynamic change of the collider shape, but provides adequate optimizations, so it is fast and efficient.

Unity provides the Mesh component, which contains data as vertices, triangles or normals. Design of this object is more oriented towards meshes used in graphical rendering. For example, it allows only indirect access to its data. For this reasons, implementation of Dynamic Mesh Controller does not use it directly, instead only the definition of the mesh shape is extracted from the Mesh component.

5.2.2 Inner data representation

The Dynamic Mesh Collider stores its data in a similar format as the Unity Mesh component. Both vertices and triangles are stored in an arrays. Vertex is represented as three dimensional vector, using Vector3 struct. Triangle is represented by array of indices which points to the vertices array. When needed, vertices, which forms the triangle, are retrieved from vertices array by accessing the positions defined by indices stored in the triangle data structure. This allows quick access to vertices which forms the triangle, either for read their value or change it, with only minimal algorithmic or memory overhead. It is also possible to add or remove vertices and triangles, by properly arranging the data inside the arrays and setting their sizes in that way, that they could contain all the data at the same time. By marking some triangles as an invalid, they can be skipped during collision calculations. Normal vectors of the triangles or not stored, as usually they are needed only for a few triangles at a time. They can be easily computed from the triangle data when requested.

5.2.3 Collision detection algorithms

For collision detection and resolution, the Dynamic Collision Mesh provides two operations:

- ray cast
- sphere-triangle intersection

Ray cast is a mechanic when virtual ray is projected from defined point (origin) to defined direction, while detecting collisions between the ray and obstacles. Optionally, maximum distance of projected ray can be set. Often only detection if something was hit or the position of first hit matters and this is the case of the implementation in the Dynamic Mesh Collider. When ray cast is requested, ray is cast against a group of triangles. If a triangle is hit, the hit position and distance are retrieved. Ray cast algorithm stores the best hit (the one with smallest distance from the ray origin) and this hit together with associated triangle is returned as a result. Möller-Trumbore intersection algorithm is used in the implementation. This algorithm is quite fast, as it uses only basic vector mathematics with few cross and dot product operations. Example of ray cast can be seen on the figure 18 (page 23). There, ray was cast from origin X against a triangle. In this case, the intersection occurs at the point I (MÖLLER & TRUMBORE, 1997).

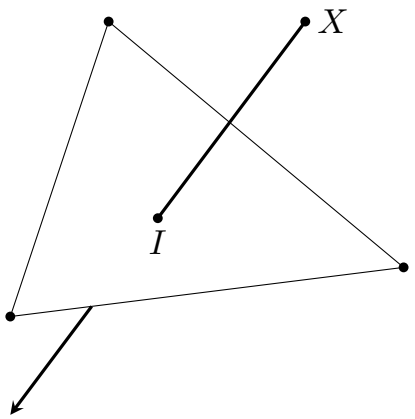

Figure 18: Ray casting against a triangle.

Sphere triangle intersection is an algorithm for detection if a triangle is intersects with a sphere. Sphere is one of the most effective shapes for collision detection, as only its center and radius are needed for calculations. The goal of the algorithm is to find a point on the triangle with the smallest distance to the center of the sphere. Collision occurs if this distance is smaller than the radius of the sphere. The problem is how to find this point, as there are number of special cases. The point can be either located on the triangle surface, triangle edge or on the triangle corner point. To find this point, implementation in the Dynamic Mesh Collider uses three phases:

- The detection against the triangle surface.
- The detection detection against the triangle edge.
- The detection detection against the triangle point.

All of them are designed to use basic mathematics and vector cross and dot product operations. The phases are ordered in that way, that if collision is detected the distance is the smallest possible and algorithm can end.

Triangle surface test is based on casting a ray from sphere center in the direction of inverted triangle normal vector. If ray hits the triangle, the hit position is also the closest point on triangle surface to the sphere center. Collision between the ray and the triangle is calculated using the previously described ray casting algorithm. Example in the figure 19 (page 24) shows the case, when sphere collides with the triangle ABC and the closest point I from the triangle to the sphere center S is on the triangle surface, but not on its edge.

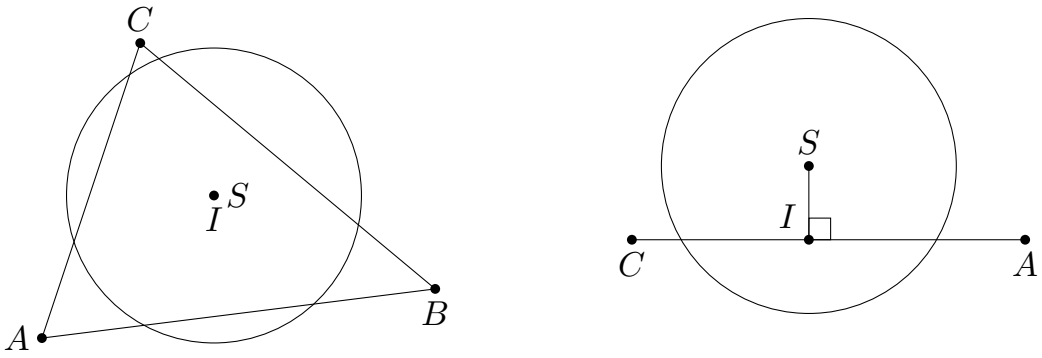

Figure 19: Collision between triangle a surface and a sphere.

Previous test will fail if the direction of the ray points out of the triangle surface. In this case, test is run against the triangle edges. The closest point on the triangle edge to the sphere center is the point, from which the line perpendicular to the edge will intersects with the sphere center. To find this point, vector projection using dot product can be used. This case is shown on the figure 20 (page 24). Closest point I from the triangle ABC to the sphere with the center S is located on the edge BC. The point I is calculated as projection of vector $\vec{BS}$ to vector $\vec{BC}$. This case shows, that if previous algorithm is used, the ray cast from point S will miss the triangle surface (shown as point Y).

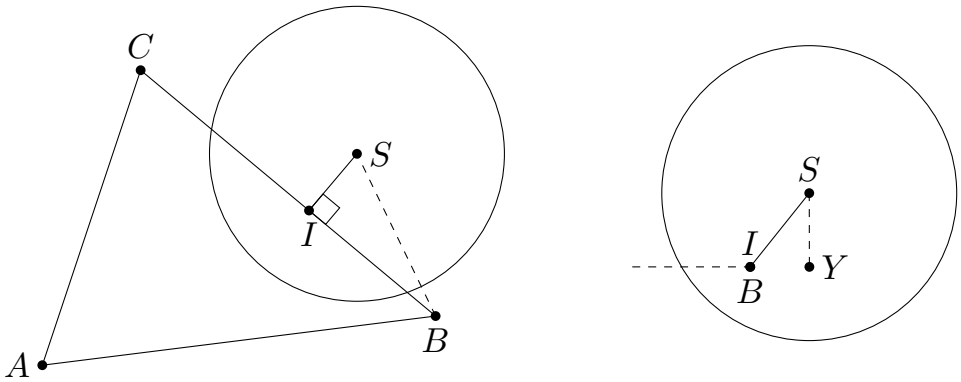

Figure 20: Collision between a triangle edge and a sphere.

Second test can fail in a scenario, when the line perpendicular to the edge and intersecting the sphere center will intersect the edge outside of the triangle. In this case, collision between sphere and triangle point can occurs. This test is the most trivial one, as it is based on calculation of distance between sphere center and triangle points. Shown on the figure 21 (page 25), the closest point I from the triangle ABC to the sphere with the center S is located on the point B. The point Y shows, that previous test algorithm by using projection of the vector $\vec{BS}$ to the vector $\vec{BC}$ will not detect collision, as the projected point Y is lies outside of the edge BC.

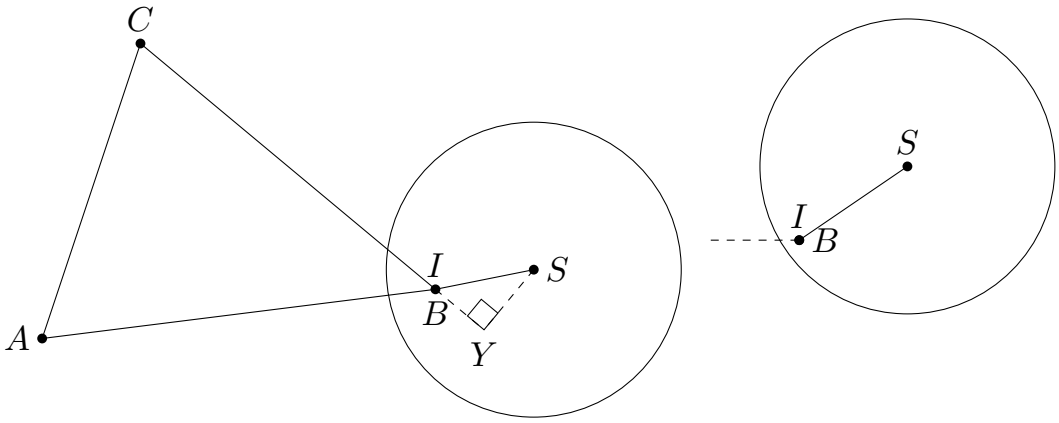

Figure 21: Collision between a triangle edge point and a sphere.

For detection if a collision does not occurs, the collision test have to fail in all phases. This is needed because the negative result in one phase does not rule out the positive result in another one. For example, the surface and edge detection will fail if the collision vector is outside of the surface or edge range. The order of the phases is also important; if collision is detected when test is run against the triangle edge point and result is positive, it does not mean that closer point on the triangle surface does not exists. On other side, if the point is detected during the test against the triangle surface, it is also the closest point to the sphere center.

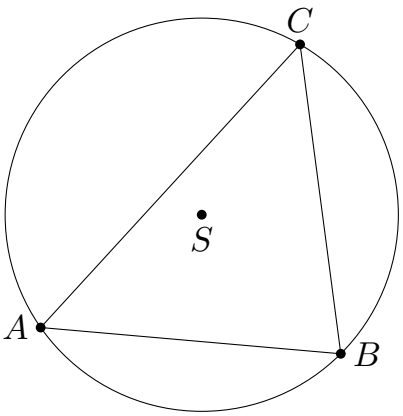

Figure 22: Triangle bounded inside a sphere.

As these operations can be too expensive when run against all the triangles from a group, it is possible to bound them inside the bounding spheres and detect collision against these spheres first. Example of

bounded triangle is seen on the figure 22 (page 25). This assumes, that the size of the triangles is similar than the size of bounding spheres, as otherwise the bounding spheres will be unnecessary large and its usefulness will be significantly lowered.

5.2.4 Collider partitioning

By using a proper data structure, it is possible to quickly detect which parts of the collider can possibly collide with the ray or another object and run the detailed collision tests only against the triangles from these parts. As the Dynamic Mesh Collider is a dynamic structure, data structure used for its partitioning should reflect this requirement. Data structures with fast modification operations are preferred in this task.

Approach used in implementation assigns every vertex of the colossus to one or more bones. These vertices are close to each other, as they forms a part of the colossus's body, as limb or head. When a bone moves, all the vertices associated with that bone will moves in the same or similar direction (because some of them can be also influenced by the other bones). As they are always close to each other and moving in a similar way, it is possible to bound them inside a primitive collider shape, and thus detect if collision with them can or can not occur.

Using this fact, vertices and triangles of the Dynamic Mesh Collider are assigned to multiple parts, named the Dynamic Mesh Collider Part. Each Dynamic Mesh Collider Part has its own bounding sphere, which bounds all the associated vertices. On the figure 23 (page 27) is shown example of these bounding volumes on one of the colossus's arms.

The bounding volumes can be either precomputed or dynamically updated on the fly. The precomputed bounding volumes are cheaper to use from algorithmic view, but they have to contain all associated vertices in every phase of colossus animation. This means, that depending on used animations, the bounding volume can be unnecessary large, thus lowering its efficiency. Another possible problem is the possibility of invalid state, when vertex will leave the bounding volume, what will probably lead to invalid collision detection.

On the other side, dynamic update adjusts the bounding volume in relation to actual position of associated vertices. The amount of vertices from which the collider is formed can be between hundreds and few thousands, depending on its precision and shape. In case of calculation of the bounding sphere and its center, two loops are needed, calculation of the center of vertices and the maximum distance to the center. The operations inside the loops are simple vector additions and scalar multiplications, so even that the bounding algorithm runs in linear time, it is quite fast. As the goal is to use inverse kinematics for colossi movements, the dynamic approach is used, as the requirements for computational power are quite small and precalculation is much more restrictive and less error prone. Also, the only shape used for the bounding volumes is the sphere shape. For some of the colossus's parts, such as limbs, the capsule collider is much more suitable, as the bounding volume can be smaller. In future, this functionality will be probably implemented.

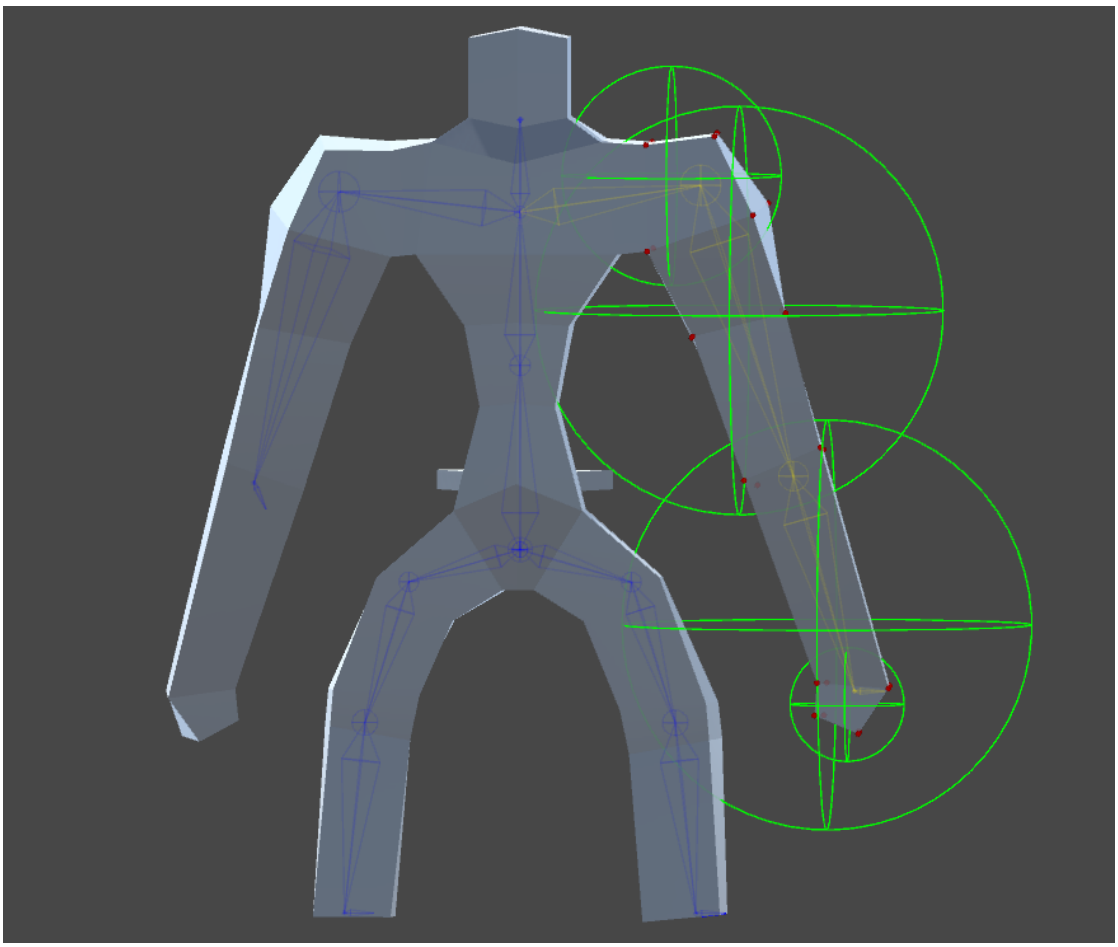

Figure 23: Dynamic Mesh Collider Parts bounding spheres of a colossus's arm.

When a collision algorithm is run (either the ray cast or the sphere collision algorithm), it is first run against the part's bounding spheres. Detailed collision test are then run only against the triangles in the parts, where collision is still possible.

To determine if bounding sphere was hit, either by a ray or an another sphere, these collision tests are used:

- sphere-sphere
- ray-sphere

In case of the sphere-sphere collision, the distance between the spheres centers subtracted by their radii defines, if and how deep the two spheres collides. On the figure 24 (page 28), the collision occurs between two spheres with centers S_1 and S_2. The distance between points I_1 and I_2, which lies on line between these two centers, defines if the collision occurs (as in this case), if the spheres touches each other, or if they are separated.

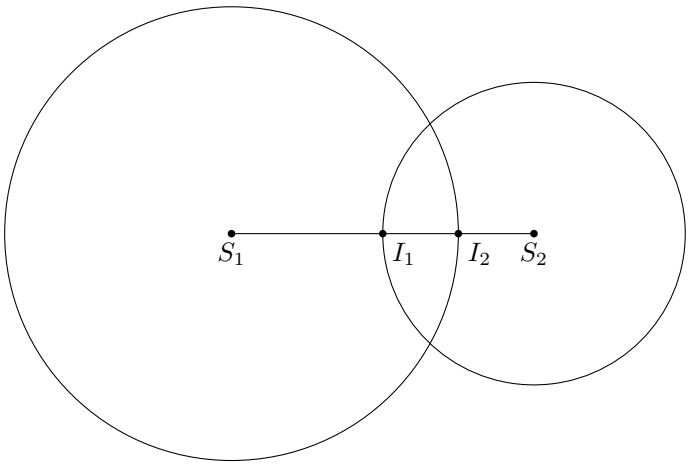

Figure 24: Collision between a two spheres.

In case of ray sphere collision, the goal is to find the closest point on the ray to the sphere center. This is possible with the vector dot product projection. Example is shown on the figure 25 (page 28), where the vector $\vec{XS}$ from origin X to the sphere center S is projected on the direction of the ray as the point I. This point is also the closest point to the ray from the sphere center S.

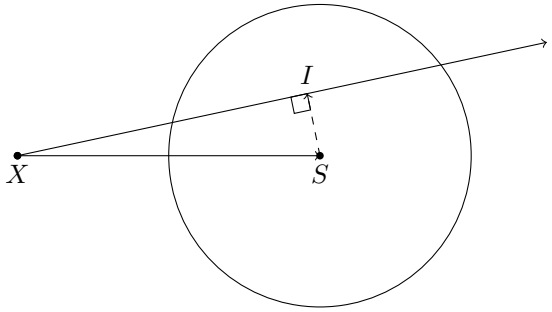

Figure 25: Collision between a ray and a sphere.

5.3 Armature and bones

The Armature script and the Bone script components are used for the animation of the Dynamic Mesh Collider shape. The skeleton of the colossus is a tree graph, formed by the empty game objects, arranged in a hierarchical structure. On the top of the hierarchy is the main bone, usually a torso or a similar main part of the body. The bones can either mimics the bones in human skeleton, or can be used for another function, like movement of vertices used in facial expressions. When a bone is moved or rotated, all child bones are transformed in regards to movement of the parent bone. For example, rotation of a shoulder bone will rotate the entire arm, but rotation a the palm bone will rotate only the palm and its child bones, as fingers. Example of how the bones forms the skeleton of the colossus can be seen on the figure 26 (page 29), where the bones, colored blue, are visible.

To properly animate Dynamic Mesh Collider, every vertex from the collider is associated with one or more of the bones and its position is stored relative to the bone local space. The bone also stores its rest

position, what is an initial position of the bone in the skeleton. When requested, the Bone Script can calculate actual positions of the vertices in regards to actual position and rotation of the bone. As the vertex can be influenced by multiple bones (in Unity, the limit is 4 bones), its final position depends on its weight parameters of the associated bones.

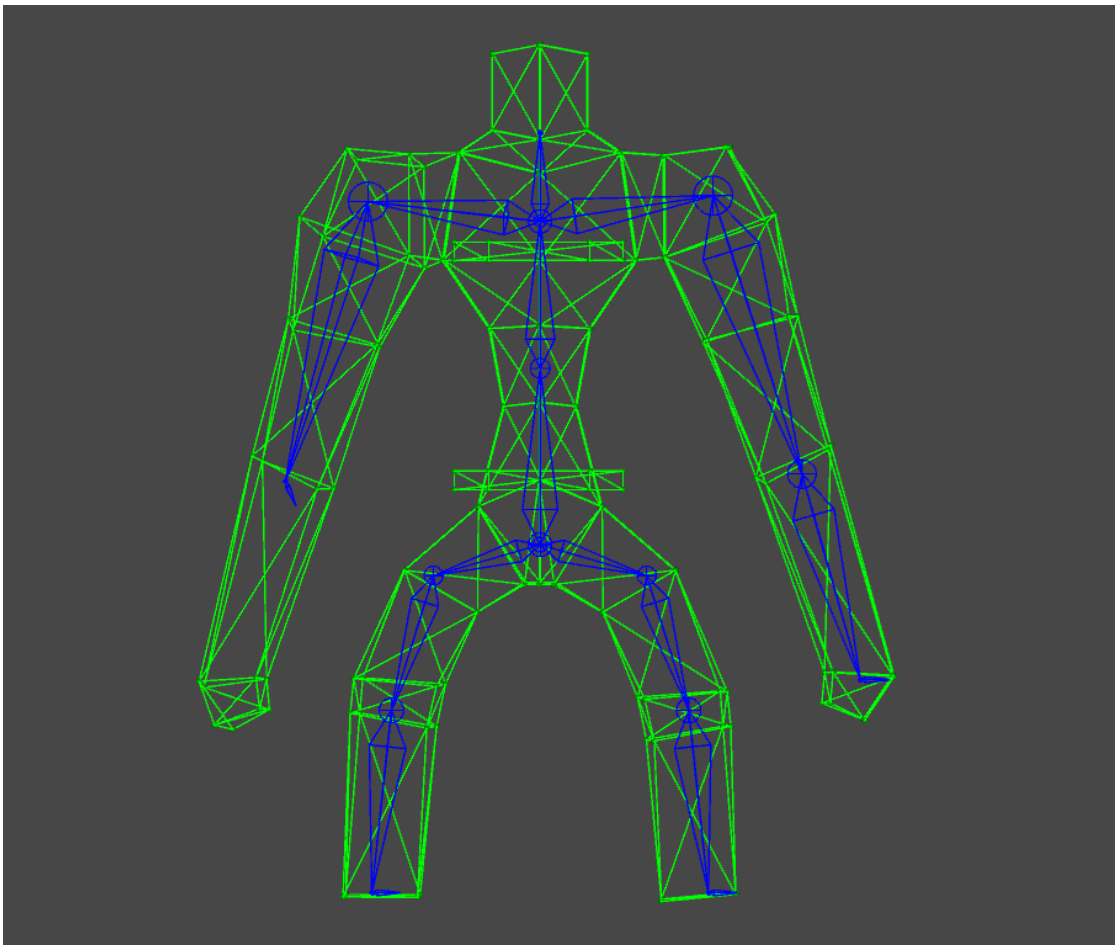

Figure 26: The mesh collider shape and the bones of the colossus.

Armature handles the update of the Colossus's Dynamic Mesh Collider shape in relation to the bones movement. It has two tasks:

- To update the collider shape.
- To update the bounding spheres of the collider parts.

Collider shape is updated in two steps: first, positions of the vertices are updated in relation to the movement of the bones, and second, the bounding volumes of the collision parts are updated in relation to new positions of the vertices.

The recalculation of vertices can be relatively expensive, as it means to iterate over every vertices of the colossus collider mesh. To further optimize this mechanics, it is possible to use one fact: the collider shape have to be updated only when the player character is near the colossus. This way, colossus can be wrapped in one large bounding volume used to detect if the player is near it. Only if this is true, the shape of the colossus collider mesh will be updated.

As the vertices count of optimized collision mesh is only between hundreds and few thousands, their update is not a performance problem for modern hardware. Also, if the shape of the colossus collider is updated only in relation to the player character, it rules out the more complex interaction between the colossus and the environment. For example, if the collider shape is constantly updated, it can collide with environment objects like rocks, trees or buildings, without usage of scripted events. For these reasons, described technique was not implemented.

5.4 Import and setup

Armature script needs data about the bones, associated vertices and their weights. Unfortunately, it is not possible to access some of these data directly. Also, some of the data should be present in the colossus game object during the work in the Unity editor. Because of these requirements, Blend Import Script component was created. Its purpose is to extract needed data from available sources and their import to game objects as the colossus or the player character.

Colossus body and skeleton used during development are created in Blender, 3D computer graphics software. The file format used by the program is the .blend format. Unity supports it directly, so the file just have to be in the Assets folder and it will be automatically imported. Imported object contains shape, materials, textures, animations and skeleton data. To prepare the colossus object from the imported data, the Blend Import Script takes these steps:

1. Initial colossus shape is baked from Skinned Mesh Renderer.
2. Baked mesh is checked for duplicate vertices and if found, they are merged.
3. Game objects representing bones are populated with the Bone Scripts and initialized.
4. Vertices are assigned to bones.
5. Collider parts are created.

As meshes in Unity are usually used for graphical representation, they have some properties which are not suitable for collision meshes. For requirements of graphical rendering, the triangles of the mesh are usually separated as this case is more suitable for graphical rendering. This means, that they do not share the vertices between each other. For collision mesh, this is not needed and actually, it brings more complications and performance overhead. For example, as one point of the space can be occupied by multiple vertices of multiple triangles, they all have to be moved when the collision mesh is deformed. If merged (so the triangles share the vertices between each other), only one vertex have to be moved. Thus, merged vertices brings less overhead, as approximately only one third of the vertices count is used in comparison to when separated triangles are used. Another problem is the inconsistency of the direction of normals between the mesh assigned directly from editor and mesh baked from the Skinned Mesh Renderer, when the normals of the mesh retrieved from Skinned Mesh Renderer are flipped. For this two reasons, Blend Import Script allows to merge the vertices which shares the same position and modify the triangles so their normals are flipped.

To assign the vertices to the bones and to create the collider parts, mapping of vertices to bones is needed. This data are not provided directly in Unity, but it is possible to extract them from Skinned Mesh Renderer. In this component, it is possible to access all bones of the colossus, read the indices of influenced vertices and create the needed mapping.

5.5 Player character

5.5.1 Movement and shape

In platforming games, one of the most important things is to have accurate and predictive control of the player character. The same applies for Shadow of the Colossus, where many aspects of platforming games are used.

In the original game, the collision shape of the player character is modeled as a sphere. This brings some disadvantages, as for example, when the player character approaches a vertical wall, it is stopped before it actually touches the wall. Because of the level design, this is not a concern (size of passable terrain or platforms is always big enough) and it brings an performance advantage when dealing with collisions, as only the test between one sphere and collider triangles have to be performed.

In this implementation, the sphere approach is used for the same reasons. It simplifies the collision tests and it is accurate enough. If there will be requirement in the future for a more accurate shape, it is still useful to bound the player character inside the sphere shape or similar bounding volume. In similar way how it was used in other components, test between collider and player's bounding shape can be run before the detailed collision tests. When collision can occur, the detailed tests are run, and only between the nearest triangles and colliders which forms the player character body.

To decide how to implement collision code for the player character, two approaches were considered:

- Custom code for both the collision with a colossus and other objects.
- Built-in collision mechanics for normal collisions and custom code for collision with the colossus.

As the collision mechanics of the colossus are outside of Unity physics module, the custom code for collision detection and resolution with the colossus is needed in both cases. In the first case, the Dynamic Mesh Collider, with some modifications, can be used for another objects, like terrain or buildings. This way, collision resolution will share the code for all objects, and the results of resolution will be consistent. The second case use different collision code for collision with colossus and other objects, so the code is more complex and results of collision resolution against various objects can be different. On other side, it allows to use built-in optimized physics when collision with other rigidbody objects occurs.

5.5.2 Collision detection and resolution

Because of the time constraints, simple discrete collision system is used in the implementation. Collision resolution occurs in two stages: detection of the collision and change of the player position in relation to the collision parameters.

The implementation detects collision between a sphere and 1 to many triangles, where the sphere represents the player character and the triangles forms the shape of the colossus. When collision is detected, the resolve vector for each collision is calculated. The results are averaged, and the player is moved to the new position. No tests to detect possible new collision are run after the player character is moved, as it will be resolved in the next frame. Example of this mechanics is shown on the figure 27 (page 32). The player character, represented by the green sphere, collides with the triangles outlined with the red color. As the weight difference between the player character and the colossus body is significant, the colossus object is treated as a static object when the player collides with it.

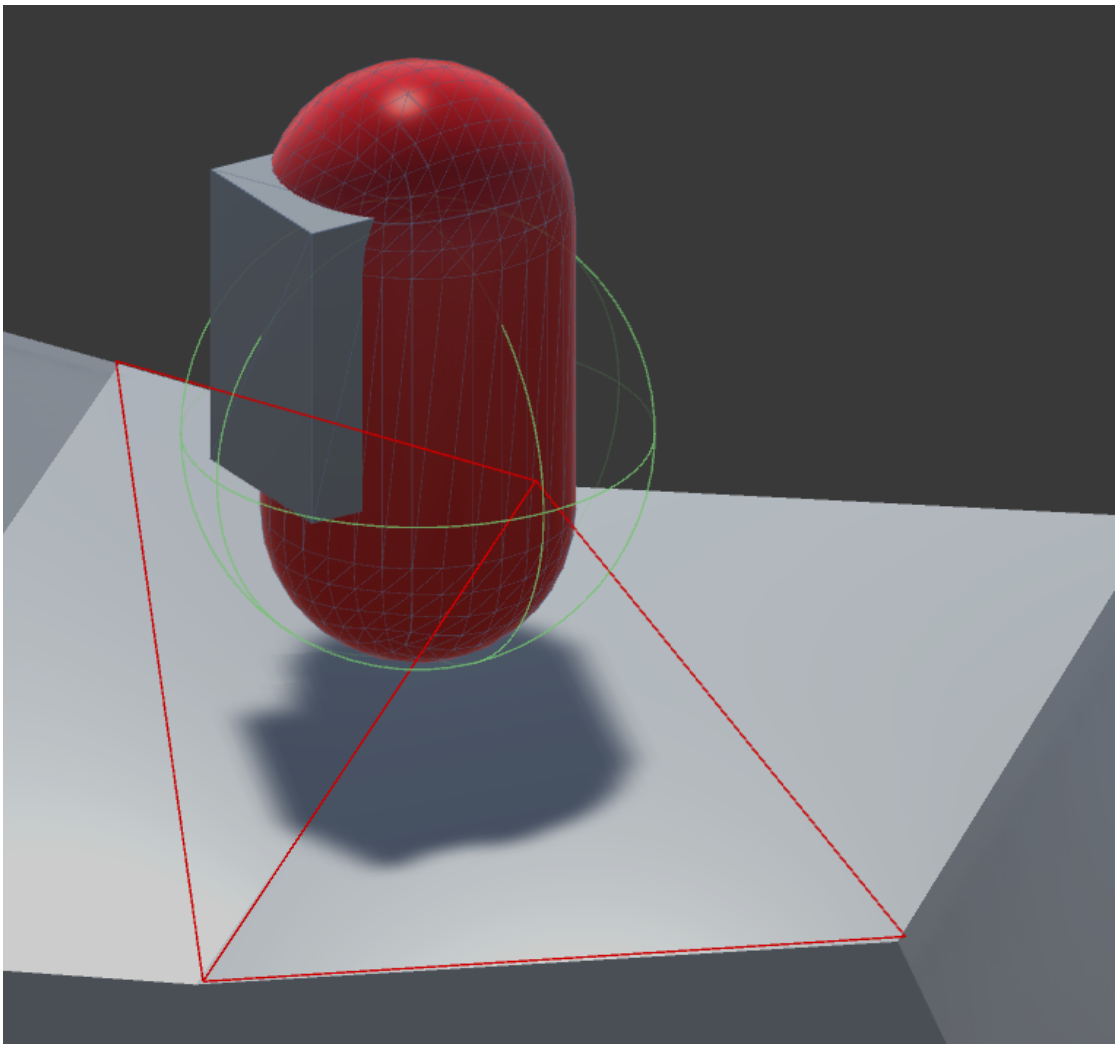

Figure 27: The player character in collision with the colossus's triangles.

5.5.3 Momentum inheritance

When the player character stands on a surface, and the surface moves, the player character should inherit the momentum of that surface. In other words, in relation to the speed of the movement and situation, the player character should keep its relative position to the surface and move together with it. This is a crucial part of the interaction with the colossus. When the player is stands on the colossus and he does not move, his relative velocity to the standing surface should be zero, even if the colossus moves. This can be achieved in two ways:

- By simulation of friction.
- By keeping the relative position of the player in relation to the surface.

In the implementation, keeping of the player character relative position to the standing surface was chosen. When the player character stands on the colossus, he touches one or multiple triangles. These touches can be represented as the contact positions on the triangles, using barycentric coordinates system. In the end of every frame, these contacts are saved together with their global position. In next frame, the new global position is calculated from the barycentric coordinate in actual triangle shape and the player character is moved according to the difference between previous and new position. This mechanic is shown on the figure 28 (page 33), where the point X located on triangle ABC is properly moved when the triangle is deformed. By converting the position of the point X from the barycentric to the world coordinates its movement can be calculated.

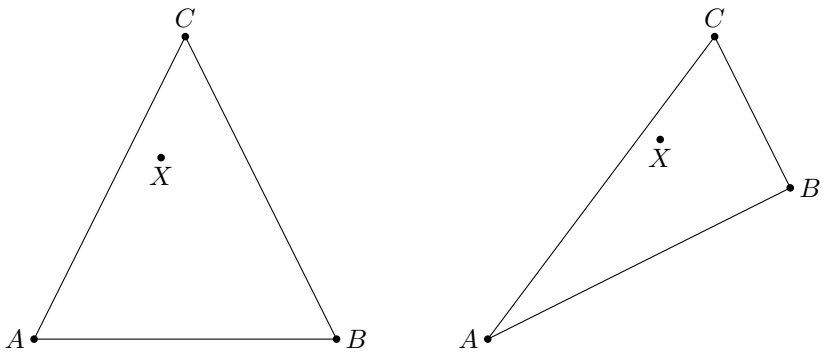

Figure 28: Relative position of the point on a triangle after its shape changed.

5.5.4 States

The motion mechanics of the player character differs in relation to its actual movement state. In original game, depending on actions available to player, these states can be observed:

- standing
- holding
- hanging
- falling

The standing state is a state, when normal movement of the player character occurs. In this state, the player stands on the ground (or colossus) and can run around, initiate a jump or execute actions like attacking with a sword or shooting arrows from a bow.

Holding state occurs, when the player crouches down and holds on something, like colossus's fur. In this state, he can still move and use some of his actions (for example, stabbing ground with a sword), but he keeps the crouched pose. Depending on the situation, like holding on a surface covered with fur, he will also climb the surface when moved. If the surface become unstable, like when colossus shakes, the player will switch to hanging state.

When player is holding on a surface (like grabbing on colossus's fur), but the surface is too unstable for a firm hold on it, hanging state is entered. In this state, the player character can not move or use his actions, except to continue the hold on the surface. The player character has limited amount of energy and this energy is consumed when he have to hold on the surface. If the energy is depleted, the hold will be released.

Falling state is activated when player character does not stand or hold on a surface, and is falling down. In this state, no actions can be performed.

The same states with similar mechanics are also used in this implementation. The state of the player is set up by state machine. The most important parameters which influences, in which state the player character is, are if the player character stands and if the movement speed of holding surface is over the limit for stable hold. Calculation of the movement speed of the holding surface is trivial, as only the position of point, where the player characters is attached is compared between the frames. The standing check is more complex, as it deals with bottom and side collision with the bounding sphere and ray casting to the bottom direction to detect, if player can stand on any surface.

5.6 Colossus AI

Various ways can be used to control the behavior of the colossus. For the needs of the implementation, simple state machine system is used. Colossus can be in various states, for example approaching the player character, waiting, attacking, etc. To determine the actual state of the colossus, the data from the sensors are used. Sensors process various data from the environment, as if the player is somewhere on the colossus, or if the player is located in specific position relative to the colossus orientation. As reaction to data from these sensors, specified state can be entered. For example, if player character is in front of the colossus, and colossus is not in attack state, the state machine will enter the attack state.

In future, another approaches can be tried, which are more complex, less predictable and more able to adapt and react to player actions. On the other side, all this mechanics should be used only if they will enrich the gameplay aspect of the game.

6 Results and Conclusion

The goal of this thesis was to replicate the gameplay mechanics of the Shadow of the Colossus video game, with main focus on the technical aspects. The main challenge was to replicate the interaction between the player character and the colossi. The technology selected for the implementation was the Unity engine.

The main tasks were identified: representation of the shape of the colossi in the collision engine using the mesh collider, interaction between the player character and the colossi as the collisions or the momentum inheritance and animation of the colossi using the bone animations.

During the process, several deficiencies were found in the components for physics and collisions handling provided by the Unity engine. These deficiencies means that it was not possible to use these components for the colossi. Main problem was the inability to use the mesh collider for the complex moving and shape changing objects.

Because of this, it was decided to built custom collision system. Another problems surfaced, but they were solved. For the colossi, custom component named Dynamic Mesh Collider, focused on providing mesh collider shape for objects deformation with associated collision detection mechanics was designed and implemented, with focus on optimized code. Support components for animating this components in sync with the bones animations and Skinned Mesh Renderer (used for deformation of the visual mesh) component were created. For import and setup, special component for parsing data from the Blender 3D editor program was implemented. Player character component for player movement and collision resolution with the colossi and other objects was designed and implemented.

Thus, the task of replication of the gameplay mechanics of the Shadow of the Colossus in the Unity engine was shown as possible, but not without problems, as the entire collision system for detection and resolution had to be designed and connected to other built-in components. In the current state of the project, the collision system is finished and works without problems. The bone animation system, designed to work in sync with the Skinned Mesh Component is also finished. The movement of the player character is still being developed, as this is one of the most important part of the gameplay aspect of every game, but from the point of the interaction with the colossus, even that not perfect, it is also working.

The entire system was designed with compatibility, efficiency and scalability with mind. The most computational heavy part, the collision detection against the colossus shape, is heavily optimized. In actual implementation, collision culling is used, when entire parts of the colossus are marked as not colliding, and the detailed collision test is run only against small portion of colossus's shape. Thanks to this, the design can be used without problems in a low performance devices, such as the mobile platforms, if the models of collision and visual shape are accordingly adjusted.

References

1. D'Angelo, 2014, *Breaking the NES for Shovel Knight*, URL (accessed *24. May 2015*): http://www.gamasutra.com/blogs/DavidDAngelo/20140625/219383/Breaking_the_NES_for_Shovel_Knight.php
2. Ueda, Sugiyama, Seki & Tanaka, 2005, *The Making of Shadow of the Colossus*, URL (accessed *24. May 2015*): http://www.scribd.com/doc/170524064/The-Making-of-Shadow-of-the-Colossus
3. Adam siddiqui, 2011, *Shadow of the Colossus HD - Valus - First Colossus*, URL (accessed *24. May 2015*): https://www.youtube.com/watch?v=GgoGa2bz__M
4. omegaevolution, 2010, *Castlevania: Lords of Shadow - Chapter 2 Boss 1: Stone Idol Titan*, URL (accessed *24. May 2015*): https://www.youtube.com/watch?v=n66iAvL0vHQ
5. OlympianDawn, 2012, *God of War 3 - Cronos Battle HD*, URL (accessed *24. May 2015*): https://www.youtube.com/watch?v=LoXRufH8tPs
6. Leadbetter, 2010, *The Making of God of War III*, URL (accessed *24. May 2015*): http://www.eurogamer.net/articles/the-making-of-god-of-war-iii
7. *Unity Physics*, URL (accessed *24. May 2015*): http://docs.unity3d.com/Manual/PhysicsSection.html
8. *Unity Mesh*, URL (accessed *24. May 2015*): http://docs.unity3d.com/Manual/class-Mesh.html
9. *Unity Collider*, URL (accessed *24. May 2015*): http://docs.unity3d.com/ScriptReference/Collider.html
10. *Unity Skinned Mesh Renderer*, URL (accessed *24. May 2015*): http://docs.unity3d.com/Manual/class-SkinnedMeshRenderer.html
11. Möller & Trumbore, 1997, *Fast, Minimum Storage Ray/Triangle Intersection* (Journal of Graphics Tools)
12. Firth, 2011, *Speculative Contacts – a continuous collision engine approach*, URL (accessed *24. May 2015*): http://www.wildbunny.co.uk/blog/2011/03/25/speculative-contacts-an-continuous-collision-engine-approach-part-1/