Game Engine Architecture Parkour

Ramy Abousaif
Department of Computer Science and Creative Technologies
University of the West of England
Coldharbour Lane
Bristol, UK
ramy2.abousaif@live.uwe.ac.uk

Abstract

Parkour systems are considered a necessity for fast paced video games as it allows for fluid and rapid movement throughout the environment to reach a certain goal. To replicate parkour in games, analyzing real life parkour moves and looking back at predecessors that implemented it, is as important as actually implementing the system. Unreal Engine and Unity Game Engine were used to attempt to implement a parkour system. Within those game engines we can create a path to a specified location via vectors/lines formed from the player's position to the location they are heading towards; And with the addition of animations and other techniques we can replicate the movement along with its fluidity. With all of that figured out, a working system with features like climbing structures, vaulting, mantling, sliding and running along walls is not as far-fetched as people might think initially.

Author Keywords

Parkour system; Unreal Engine; Unity game engine; Game engine parkour; Climbing system; Vaulting system

Introduction

Parkour has always been present in almost any video game, whether it is a 2D game where you jump off of platforms and walls or a 3D game focused on traversing through obstacles with optimal speed while keeping your momentum and giving the player the feeling of fluidity and awe-inspiring movement. Parkour in games might seem incredibly complex at first but with further analysis of other examples, it could be broken down and simplified. To implement a parkour system, Unreal Engine and Unity Game Engine were used (Game engines are complex systems that are formed of subsystems to provide a lot of functionality to make games).

Background & Research

Real life examples:

Looking at real life instances of others doing parkour was important to understand the foundation of it and how it works. Studying movements such as a "Speed Vault" would help replicate it into a virtual world. (Parkour Wiki, n.d.) states that speed vaults are done by placing one hand and one leg on an object to quickly get over said object.

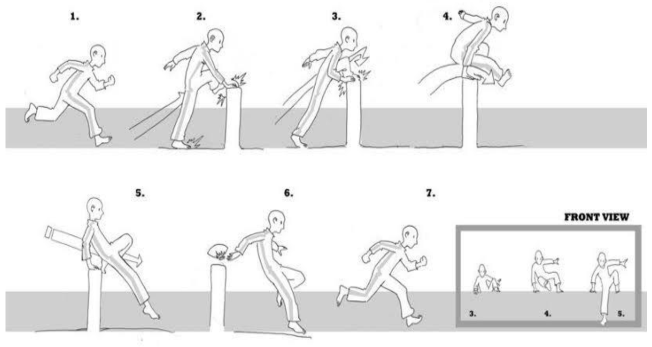


Figure 1. Visualization of a Speed Vault

Another move from (Parkour Wiki, n.d.) that was looked into is called the "Wall Climb" or "Mantling". The person performing this move would require some decent upper body strength. It involves getting a grip on the edge of a higher platform while also supporting the rest of the body by placing both feet on the wall. Then the person performing it would simply use both hands and feet to propel himself/herself upwards and over the edge.

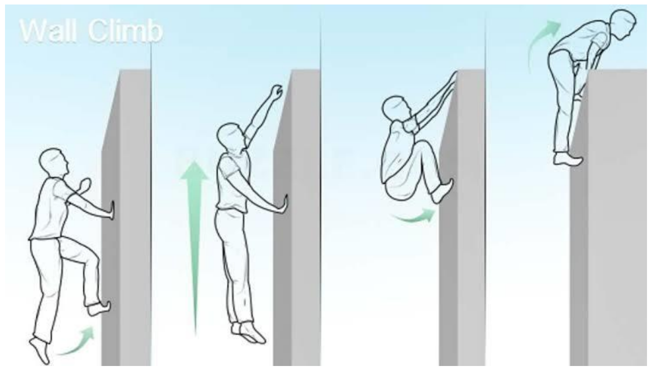


Figure 2. Visualization of a wall climb/mantle

Credit to (TappBrothers, 2015) for the screenshot and the explanation to how people wall-run

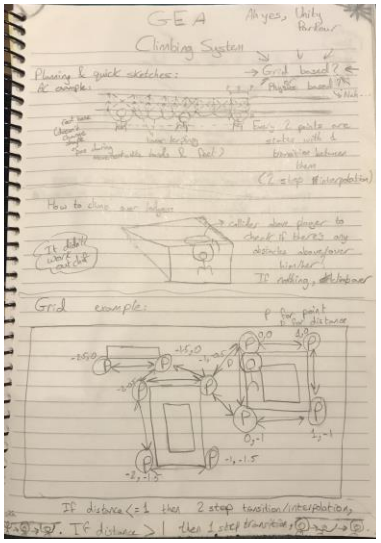


Figure 5. Sketch of a plan that shows how the principals of the system works in terms of climbing to different points and climbing over a ledge.

Wall-running is the idea of horizontally running along a flat wall in order to get past an obstacle underneath or to gain a bit more height and jump off to reach a higher platform. This technique is all about getting the right angle, having enough forward momentum and friction.

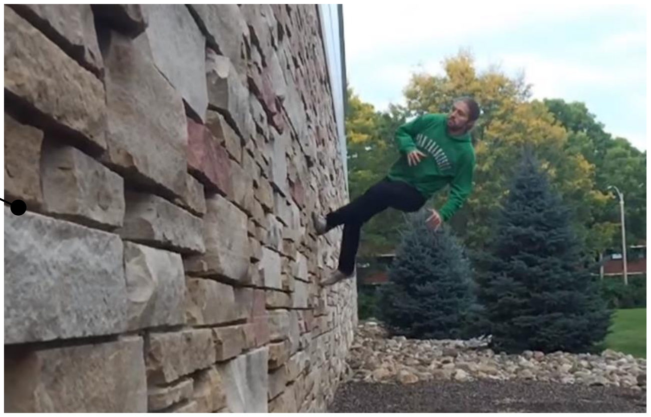


Figure 3. A man seemingly defying gravity and running on a flat wall, this technique is called a wall-run.

Last but certainly not least, wall climbing. Where a person climbs up a wall with crevasses to reach a higher point. Commonly associated with rock climbing or Buildering (A term for climbing up high buildings). When it comes to this technique it is mainly about proper foot placement. (Eric Karlsson Bouldering, 2016), a rock climbing coach, and websites like (REI.com, n.d.) state that it is better to focus pressure more on the toes (specifically the big toe) rather than the bottom of the feet to maintain stability and to gain more reach. With observing climbers, it was also clear that in order for them to maintain balance, they would start by placing their hand first and then they would place the foot on the same side as that hand second. Then rinse and repeat.

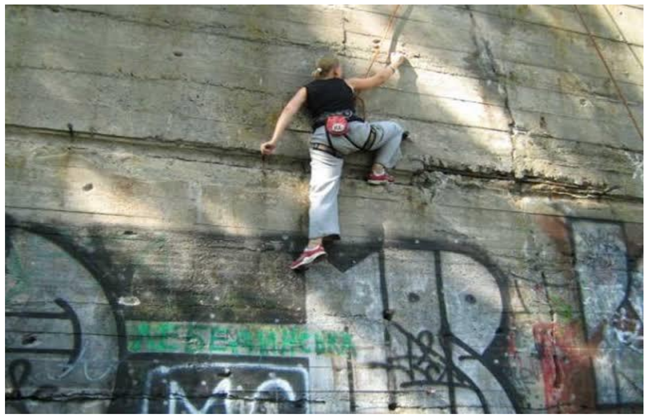


Figure 4. A person using crevasses to scale up a building (Buildering).

Examples in games:
A few games that featured parkour systems were analyzed to further understand how to implement a similar system. First game that comes to mind is "Assassin's Creed". A game known for its innovative climbing system. The game was made by a huge company (Ubisoft) with hundreds of animators and programmers so obviously it is almost impossible for one person to achieve a fluid system like theirs but there were patterns visible in their game that can give individuals a better understanding of how the system works. Surprisingly, another example was a game made by the same company, a game called "Splinter Cell: Blacklist". It shares the same elements as Assassin's Creed (since it is from the same company which probably shares resources among its teams), but the game is much smaller compared to it, so in a way it was easier to analyze it for more information about how the system works.

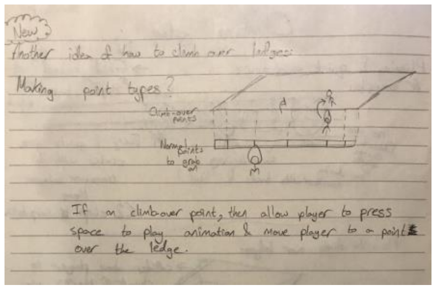

New
Another idea of how to climb over ledges:
Making point types?
Climb-over points
Normal points to grab on
If on climbover point, then allow player to press space to play animation & move player to a point over the ledge.

Figure 6. Sketch of another idea of how climbing over ledges could be handled.

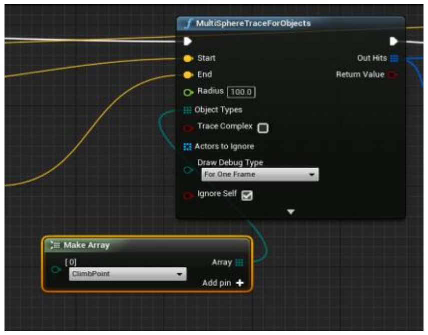


Figure 8. Blueprint code that forms the circle and specifically detects the objects that are considered climb points.

Analysis:
In these games when the player climbs up a ledge, he/she goes into an idle state on a point along the ledge. A pattern can be instantly noticed once the player starts moving along the ledge. While moving along the ledge, the player goes onto the same exact positions every time he/she goes back and forth. This gives the idea that there are already fixed positions along the ledge where the player can only hang on. From then on, the player moves to another position along the ledge, this method could be achieved by linearly moving the player's root (Essentially a point in the player's body that determines its exact position, a center of mass in a sense). This is all for climbing up ledges and other places the player would hang on a wall. As for normal ground movement, in recent Assassin's Creed games there is a system called "Free-run" which would allow the player to go from a walking/running state into a sprinting state. In that state the player can automatically vault over any obstacle in front of him/her, climb up short walls and slide under obstacles.

Implementations

The subsystems to be implemented for the entire parkour system consist of:

- A climbing system.
- A wall-running system.
- A free-running system (vaulting, mantling, sprinting and sliding).

Luckily both Unreal Engine and Unity Game Engine offered its users the ability to form something called traces. Traces are essentially lines/vectors drawn from two points. These points in this system will usually consist of the player's position and the impact location of the line (and an optional offset added/subtracted to any of the points).

Unreal Engine Implementation

Climbing system:

This climbing system was made following a series of tutorials by (CodeLikeMe, 2019). Blueprints (A form of visualized coding) in Unreal allowed for a way to dynamically find places to hang on (climb points) according to the player's location.

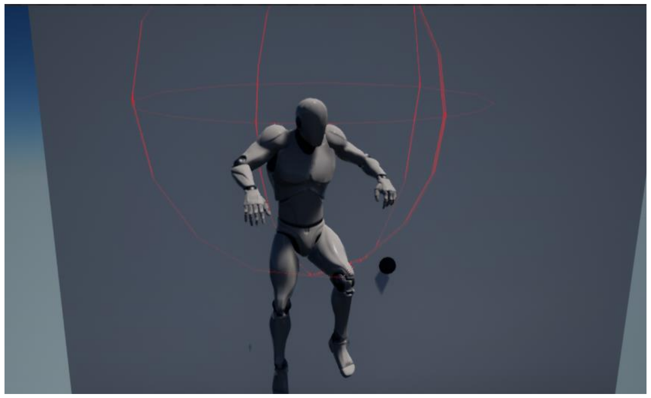


Figure 7. Player with a red sphere trace looking for any climb points on a wall.

Every time the player presses an input to jump up, a sphere trace forms around the player's upper body area which looks for any object that is considered a climb point. When a climb point is not found there is a line trace that is always active that attempts to detect a wall in front of the player. Once the line trace hits a wall another line trace becomes active that is placed a few inches in front of the player that detects how high the wall is compared to the player. If the wall is too high the player cannot climb on the edge at the top, if it is a few inches above the player, the player can hang on the edge. Since normals are used, it allows the player to move on curved walls as well.

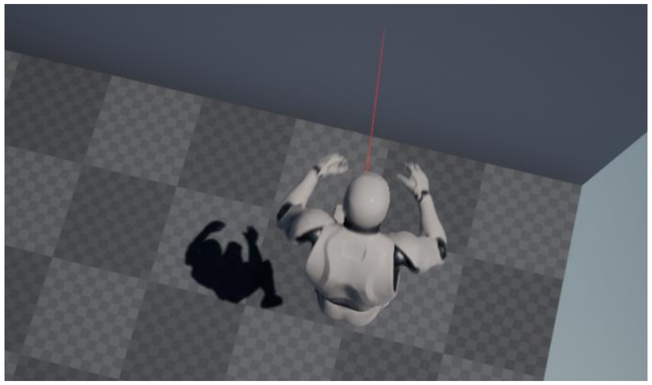


Figure 9. Player with a red line trace looking for a wall in front of him/her.

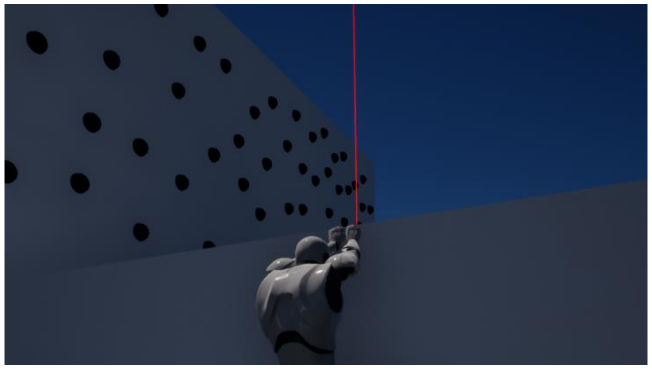


Figure 10. Player with a red line trace perpendicular to the wall to see how high the wall is compared to the player to determine whether the player can hang on or not.

The player transitions to the hanging location with the "Move component to" node in the blueprint code that essentially moves the player from point A (location where the jump was initiated) to point B (the impact location of the line trace from figure 8).

Now if the player finds a climb point instead of a ledge, the player hangs on to that point (that is closest to the player) as if he/she were hanging on to the ledge with the "move component to" mentioned from before. From that point the player's inputs are required to move the player to other climb points on the wall. This was done via another sphere trace that is a few inches in front of the player's upper body part. This time the sphere trace requires an offset that corresponds with the player's input. So, if the player wants to go left, the sphere would form on the left and once it detects a climb point the "move component to" node is used again to move the player to the closest detected climb point.

The closest climb point is determined by comparing the distance of each climb point in a looped array and the one with the shortest distance is the one the player hangs on.

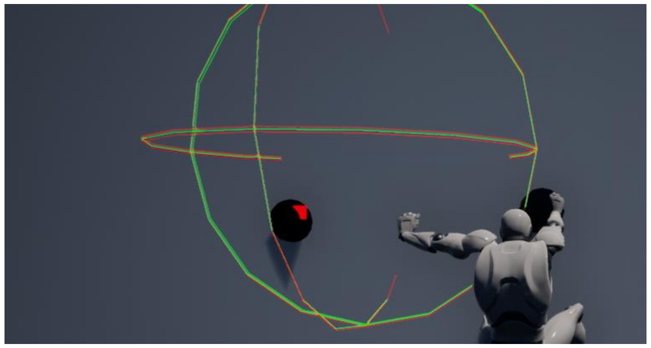


Figure 11. Player hanging on a climb point, trying to reach out for another climb point that was detected via the sphere trace.

To give the illusion that the player moves to ledges and hanging points like a real person, animations and Inverse Kinematic bones were implemented in this system. Inverse Kinematics (IK) is another way, where the movement of the chain is determined by a "target" point. Animations were mainly taken from the website Mixamo.com. IK bones were generated for the model of the player and are then assigned in various ways to their “target” points. The “target” points for the leg IK were assigned via their impact points along the wall. To put it simply, the feet positions were placed on the wall with more line traces that are perpendicular to the wall with an offset from the model’s pelvis that is defined by a certain value that can be changed according to whoever uses this system. So, for climb points the value for the leg positions would be an offset different to the offset that would be set for the legs along a ledge.

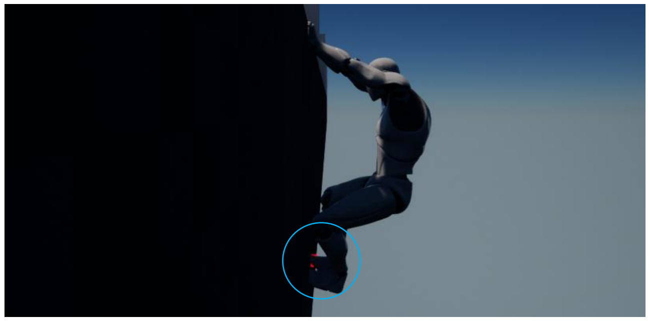


Figure 12. Demonstrates the line traces emitting from the player’s feet to the wall to adjust the Bone IK for the feet.

As for the hand IK bones, they are placed exactly on the sockets (points within the climb point) that were manually made in the climb points, therefore if the person uses this system and wants to create different shaped platforms, then he/she must place the sockets manually (Important to name them “RH” for right hand and “LH” for left hand) for the object’s static mesh.

Static meshes is essentially an object’s 3d model along with its collisions

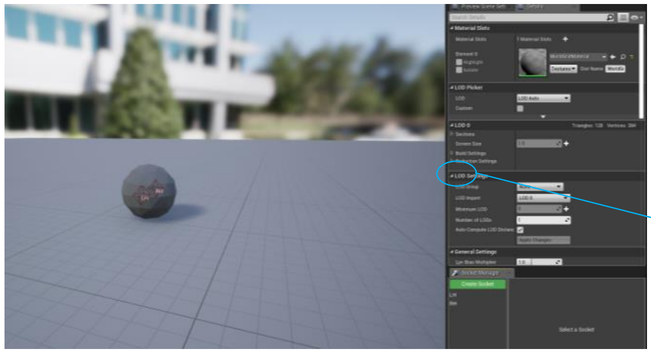

Sockets that would be manually added and placed in the viewport.

Figure 13. Shows the interface and viewport when editing a static mesh, this is where the person using this system would add the sockets.

Now if the person using this system wants to add more climbing points, there is a specific way to do so to make it easier to place the climb points in the virtual world. First the user must create the static mesh with the sockets in place. After that the user must go into “Foliage” mode and drag the newly added static mesh into the specified location that Unreal dictates. Once the user does that, the user must also add a custom made “Blueprint Class” called “FL_Hanger” to the component class in the foliage tab. The user must then change the “Collision Preset” in the foliage tab to “ClimbPoint_Collision”, this is done so that the sphere traces can detect those climb points. Finally, the user must then change the “Ground Slope Angle” in the foliage tab so that it has a minimum of zero to a maximum of 180 degrees. After that the user can essentially paint on the climb points anywhere on the

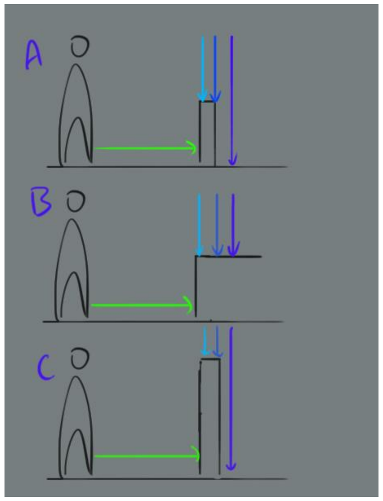


Figure 14. Visualization of the free-run system

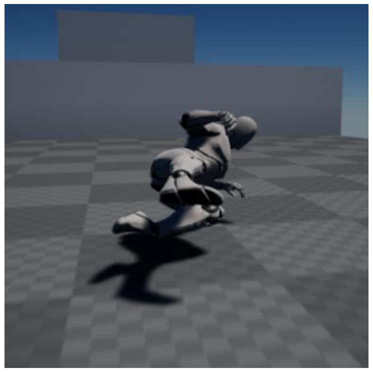


Figure 15. Player sliding

map. Although this process is long it ensures that placing the climb points in the virtual world would be easy and quick.

Free-run system:

While the player is sprinting (via "shift" key) there will be a line trace forming in front of the player similar to the one in figure 7. While this line trace is active it will search for any walls and once a wall is detected an array of other line traces (3 line traces) will form above that wall. This is all similar to the ledge climbing traces but instead of one line trace forming perpendicular and above the wall, 3 will form. Each line trace perpendicular dictates what type of wall the wall in front is. If 2 perpendicular line traces hit a low wall, that means the player can "speed vault" over it. If those 2 perpendicular line traces hit a higher wall, the player mantles the wall quickly to maintain momentum. If all 3 perpendicular line traces hit a low wall/elevated floor, the player will step on it as if it was a staircase. (See figure 14). Finally, the sliding mechanic. When the player presses the "E" key while reaching max running speed (not sprinting) the player slides down. This method is attained by lowering the player's capsule collider over a period of time - to emulate the transition from running to sliding – so that in effect the player can avoid overhead obstacles. Sliding also propels the player forward with a specified amount of force that the user can input.

Wall-running system:

This system was made following (Reid Trehame, 2019)'s explanation and mathematical calculations. This is set up with a trigger volume around the player in the shape of a capsule, when this volume is triggered by touching a wall and while the player has momentum in the air, wall-running commences. The player must also be holding certain keys to start wall-running as well. For instance, if the player has a wall to his/her right, the player must be holding the "D" key and the "W" key which are the inputs for right and forward respectively. For the left wall instead of holding the "D" key, the player must hold the "A" key. Once the player touches the wall, the system calculates which side of the wall the player is running along by using a dot product of the player's right vector and the normal of the wall (taken from another line trace that emits from the player). With that knowledge we can do a cross product of the side and the wall normal to get a vector parallel to the wall. That vector is our direction. Now that the direction is calculated, it is multiplied with the player's current speed. Gravity is set to zero and now the player can walk horizontally along the wall with a slight tilt and animation to tie it up nicely. Due to the use of normals, the player can also wall-run on curved walls.

[illegible] NEEDS TO BE REBUILT (1 unbuilt object)

Figure 16. Shows player wall-running

Unity Implementation

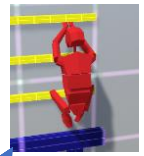
Climbing system:

This climbing system was made following a series of tutorials created by (Sharp Accent, 2016) who also supplied the assets (models and drawn curves) and animations.

This system is the equivalent of what is called a "Grid-based system". Where you place all the climb points on one plane/wall. All the points are connected to form a network of paths that the player can take, but they must be within a certain range. If they are out of range, they will not be connected to the rest of the network and the player will not be able to go towards the points out of range once he/she is in that network.

The grid is populated with points via a script that creates points on a mesh. By pressing the "create indicators" button, 2 points will be formed on the furthest left and furthest right of said mesh. The user must adjust the 2 points manually on each mesh and once they do that, they can add more points between the 2 points with another click of a button

Figure 18. Player climbing idle state



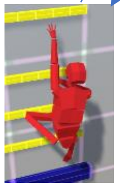


Figure 19. Player climbing, "in between" state

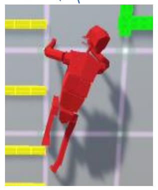
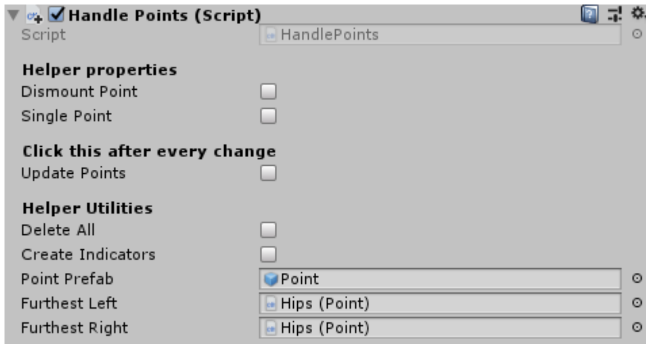


Figure 20. Player jumping from one point to another, "direct" state



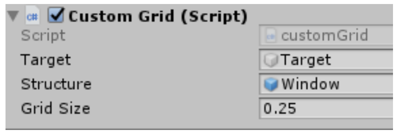
Handle Points (Script)

Script	HandlePoints
Helper properties	
Dismount Point	☐
Single Point	☐
Click this after every change	
Update Points	☐
Helper Utilities	
Delete All	☐
Create Indicators	☐
Point Prefab	Point
Furthest Left	Hips (Point)
Furthest Right	Hips (Point)

Figure 17. shows the script used to create, update and delete the points on a mesh. Create indicators will create the furthest left and furthest right points, update points will add more points between the furthest points.

The person using the system can also set the climb point's type. For example, the user can make a normal point into a dismount point by clicking on the check box next to "Dismount Point" in figure 15. Once the user does that, the player can climb up to that point and they can then climb over it and onto a suitable platform.

In this system there are two types of connections between the climb points. A 2-step interpolation/" in between" connection and a 1-step/" direct" connection. In the direct connection the player moves from one point to another with 1 swift jump. This usually occurs when the connections are past a certain (customizable) threshold in terms of distance. Then there is the 2-step interpolation connection. For example, when the player presses an input to go right once while they are on a point connected to another close point, the player will move their arm towards that point with the option to put that arm back to the current point by pressing left



Custom Grid (Script)

Script	customGrid
Target	Target
Structure	Window
Grid Size	0.25

Figure 21. The user would place whichever mesh he/she would want in the “structure” component and the grid size would dictate the intervals the object would move with. This system was made by following (Simon Holmqvist, 2018)’s tutorial.

or fully commit and jump to the other point by pressing right again.

After some consideration, a snap-to-grid system was added to make it easier for users to place gameobjects in their virtual environment. This system uses an empty gameobject called “Target” and assigns the gameobject’s position to the target’s position. The target can only move at intervals that the user can specify. (see figure 21)

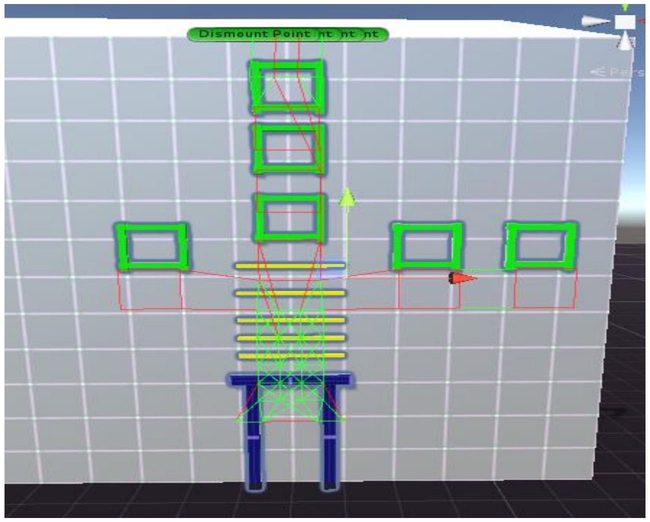
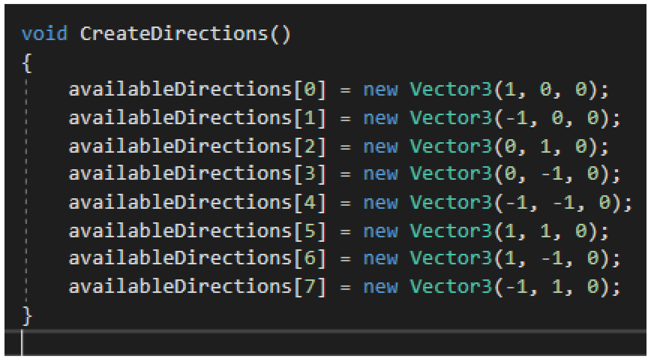


Figure 22. Shows network of paths the player can take on the grid-based system, green lines indicate that the player can do a 2-step interpolation movement, red indicates that the player can do a 1-step interpolation movement.

Just like the Unreal Engine system, the player can move in 8 directions. Up, down, left, right and diagonally in between each of them.



```
void CreateDirections()
{
    availableDirections[0] = new Vector3(1, 0, 0);
    availableDirections[1] = new Vector3(-1, 0, 0);
    availableDirections[2] = new Vector3(0, 1, 0);
    availableDirections[3] = new Vector3(0, -1, 0);
    availableDirections[4] = new Vector3(-1, -1, 0);
    availableDirections[5] = new Vector3(1, 1, 0);
    availableDirections[6] = new Vector3(1, -1, 0);
    availableDirections[7] = new Vector3(-1, 1, 0);
}
```

Figure 23. All available directions are local on each point.

The system checks the available directions on each point on the grid before runtime (running the game). Once that happens it checks if the directions are valid by comparing the angle of the direction available from each point close to the point the player is currently on. If the angle falls in a certain range than it determines that the direction is valid. Once it is valid the point is then assigned to a target point that the player can jump towards.

Closest neighbor to our current point is determined similarly to the Unreal Engine counterpart.

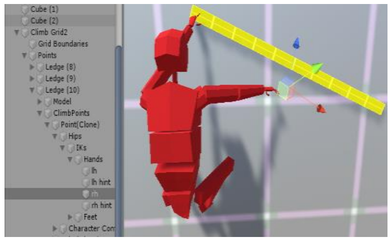


Figure 25. Before user adjusts "RH" IK.

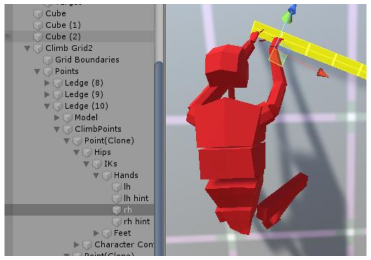
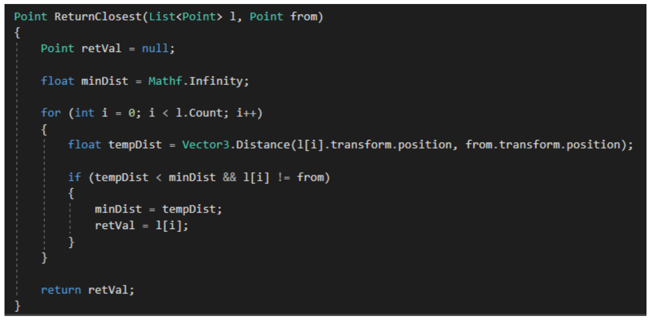


Figure 26. After user adjusts "RH" IK.



```
Point ReturnClosest(List<Point> l, Point from)
{
    Point retVal = null;

    float minDist = Mathf.Infinity;

    for (int i = 0; i < l.Count; i++)
    {
        float tempDist = Vector3.Distance(l[i].transform.position, from.transform.position);

        if (tempDist < minDist && l[i] != from)
        {
            minDist = tempDist;
            retVal = l[i];
        }
    }

    return retVal;
}
```

Figure 24. Checks closest point from the current point. Gets distance of each point compared to the point the player is on. Then checks for the closest distance while ignoring the point the player is on and returns the closest point.

Once closest neighbor is obtained it gets added to a list of neighbors and that is how connections are formed.

In this system, movement is handled partly different to the Unreal counterpart. In principal it still interpolates the character's starting position to the target position but in this system, the target position and the start position constantly change along pre-made curves. An array of points is set on a specified curve that is called according to specified function. For example, while the player is on a climb point and they decide to jump up to another climb point, a pre-made curve that corresponds to that jump will be formed and a certain amount of points will be formed along that curve. The player will move in between those points in said curve to simulate a realistic jump between the 2 climb points.

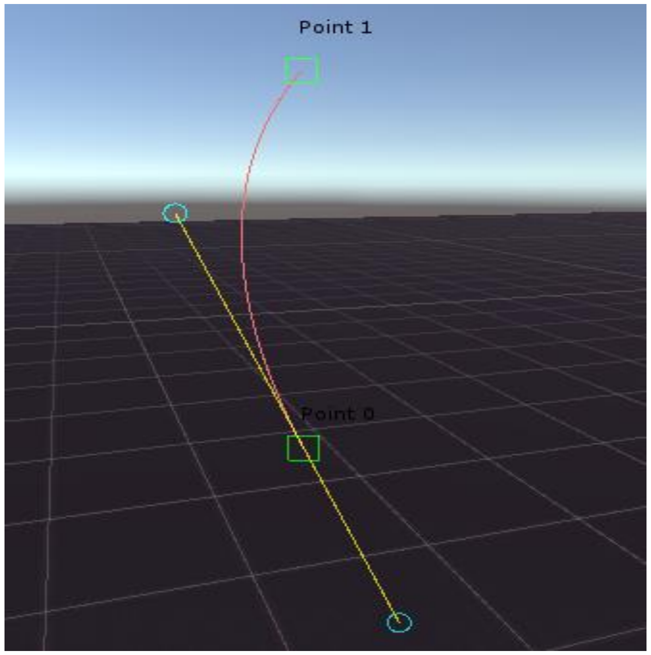


Figure 27. Example of a pre-made curve.

This system also has animations and IK bones to simulate realistic climbing. IK is almost as identical as the Unreal Engine counterpart with the exception that each point's IK bone placements are done individually. Meaning that when you go into each point in Unity's inspector, you can adjust the positions of "LH" and "RH" (left hand and right hand) gameobjects to suit the mesh you would like to climb on. (See figure 25/26).

Gameobject is a term for objects present in the game/viewport

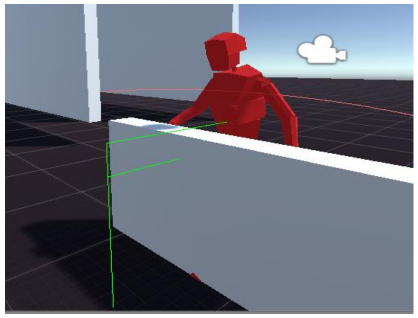


Figure 28. Traces for vaulting.

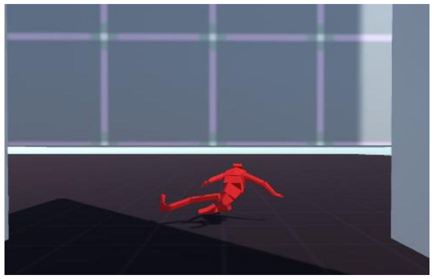


Figure 29. Player sliding underneath an obstacle

Free-run system:

While player is sprinting (holding "shift" key), the player enters a free-run state. In this state the player can run into any short walls so he/she can vault over. This was done through 3 line traces. First line trace emits from the player and goes forward. Once that trace hits an object, the second trace is a certain distance above and parallel to the first trace to detect if there is anything blocking the short wall the first trace would hit for example. If there is something blocking it for example the wall is not actually short but a tall wall, then nothing happens. If it does not hit anything the third trace is formed downwards towards the ground. If the third trace hits the ground, the player will do a "speed vault" over the short wall while also correcting the player's rotation and position to be facing perpendicular to the wall once he/she starts vaulting. The player's position is then moved to the hit position from where the first trace ends, which Is about where the short wall's edge would be with an added offset to ensure that the player does not get stuck inside the short wall.

Wall-running system:

This system only consists of 2 line traces. One that originates from the player to the right side and one that originates from the player to the left side. Since the player is properly affected by real world forces like gravity (as the player consists of a "rigidbody" component), the wall-run was made more realistic than the Unreal Engine counterpart where the player goes in a curve-live trajectory along the wall and requires to move at a good angle before wall-running. To achieve this curve-like trajectory while wall-running, the player's gravity is set to 0 once initiating the wall-run carrying upward and forward momentum. As the player move horizontally at a fixed rate, the gravity applied on the player decreases more and more until it reaches -9.81. The player can jump off the walls with spacebar to propel themselves away from the wall.

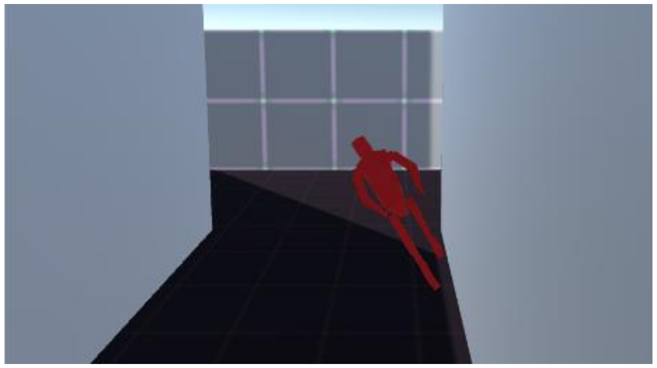


Figure 30. Player wall-running

Evaluation

What went well? What not so well? What could be improved?

Both engines allowed for a fairly smooth and simple procedure when it comes to programming the movement. Although animations and the transition between them are not ideally smooth. Detecting climb points in the climbing system has some bugs/errors (like skipping closer climb points in both implementations) that makes the player's freedom feel more limited and restricted. In the Unreal engine version of the system the player cannot move for a fraction of a second once he/she starts climbing.

How does each engine weigh up to implement this effect / aspect?

Both engines were very efficient to implement the system in, they both had similar functionalities, but Unreal Engine does have a great functionality that adds more complex traces (in the sense where it can form multiple lines at once to form the shape of other simple shapes like spheres, capsules and boxes). Because of that it allowed for a more dynamic way to detect objects like climb points.

How would this work as part of a wider game project?

Since this system has a lot of customizability like adjusting trace sizes/distances, speed, the ability to adapt to the player size automatically and the ability to adjust IK positions, it should work fairly well with any other big game project. The only issue present is with the Unity system where the player would have to make some changes to the character controller to accommodate the climbing system, but it is not a huge issue and these changes are quite easy as it is only about importing the scripts present within the system.

Compare and contrast between the engines used.

Since Unreal Engine had more complex shaped traces than Unity, it allowed the system to have a more dynamic approach to detecting climb points unlike Unity which had its climb points positions pre-determined before the game even ran. In the wall-running system, Unreal also allowed its users to use local physical constraints, which would make it so that horizontal/gravity defying wall-runs are possible, by default unlike Unity which had only world-space physical constraints available. Unity allowed for a more realistic approach to parkour as the player had a rigid body component (a component that makes a gameobject influenced by real world forces like gravity), meanwhile Unreal had a character controller with built-in features like gravity. Blueprint coding in Unreal allowed for an easier to understand interface compared to the C# scripting used for Unity, because of that it was easier to visualize mathematical calculations in Unreal, therefore coding was more efficient in terms of time consumption. Unity was especially useful for animations as it allows users to create animations from scratch, to create animations for Unreal, the user would need to use a third-party program. Since the system in Unity consists of pre-built climb points, the more points the user adds to the game, the more the load on the system will get, although that number should theoretically be incredibly high to affect performance.

Acknowledgements

Thanks to the staff who created the template for this document. Thanks to Ubisoft Montreal and Gameloft for developing franchises that ended up being the sources of inspiration for the climbing, wall-running and free-running systems in this report.

References

(n.d.). Retrieved from Parkour Wiki: https://parkour.fandom.com/wiki/Climb_Up

(n.d.). Retrieved from Parkour Wiki: https://parkour.fandom.com/wiki/Speed_Vault

(n.d.). Retrieved from REI.com: https://www.rei.com/learn/expert-advice/climbing-techniques.html

CodeLikeMe. (2019, August 17). Retrieved from https://www.youtube.com/watch?v=A42S04n5ipY

Eric Karlsson Bouldering. (2016, April 1). Retrieved from https://www.youtube.com/watch?v=oqd39cc0HL0

Reid Trehame. (2019, November 14). Retrieved from https://www.youtube.com/watch?v=B3fznaHVIMs

Sharp Accent. (2016, Septermber 25). *Unity Tutorial Climb System Series*. Retrieved from https://www.youtube.com/watch?v=k8kZ3-K8gcs

Simon Holmqvist. (2018, May 30). Retrieved from https://www.youtube.com/watch?v=eUFwxK9Z9aw

TappBrothers. (2015, November 4). Retrieved from https://www.youtube.com/watch?v=bgyrrjmavy4&t=119s