UNIVERSITY OF CALIFORNIA

Los Angeles

Data-driven Locomotion of Virtual Humans in Unity

A thesis submitted in partial satisfaction

of the requirements for the degree Master of Science

in Computer Science

by

Ankit Arora

2011

The thesis of Ankit Arora is approved.

_____

Petros Faloutsos

_____

Glenn Reinman

_____

Demetri Terzopoulos, Committee Chair

University of California, Los Angeles

2011

TABLE OF CONTENTS

1 INTRODUCTION.....1

1.1 PROCEDURAL ANIMATION.....4

1.2 PHYSICS-BASED TECHNIQUES.....5

1.3 EXAMPLE-BASED MOTION.....5

2 RELATED WORK.....7

3 OUR WORK.....10

3.1 GAME ENGINE SELECTION.....10

3.2 MOTION CLIPS PREPARATION IN 3DS MAX AND IMPORT IN UNITY.....11

3.3 DATA-DRIVEN LOCOMOTIOM MODULE.....13

4 RESULTS.....16

5 APPENDIX – SOURCE CODE LISTING.....18

6 REFERENCES.....34

LIST OF FIGURES

FIGURE 1 ARTICULATED FIGURE WITH THIRTY DEGREES OF FREEDOM...2

FIGURE 2 CHARACTERISTIC PHASES OF THE WALKING MOTION..... ..3

FIGURE 3 SYSTEM OVERVIEW AND MAIN COMPONENTS.....9

FIGURE 4 (a) CHARACTER AT WORLD ORIGIN.....12

FIGURE 4 (b) CHARACTER AFTER OFFSET.....12

FIGURE 5 ANIMATION CLIPS USED FOR THE CHARACTER.....13

FIGURE 6 (a) CHARACTER APPROACHING IT'S TARGET.....16

FIGURE 6 (b) CHARACTER AT IT'S TARGET POSITION.....16

FIGURE 7 (a) LARGE TURNING ANGLE.....17

FIGURE 7 (b) SMALL TURNING ANGLE.....17

TABLE 1 MOTION DATABASE.....7

ABSTRACT OF THE THESIS

Data-driven Locomotion of

Virtual Humans in Unity

by

Ankit Arora

Master of Science in Computer Science

University of California, Los Angeles, 2011

Professor Demetri Terzopoulos, Chair

Virtual characters are graphical analogues of real-life people capable of performing human-like behaviors. They are found in an array of fields now-a-days such as movies, games and as tutoring guides. Much research has been done in the areas of animation, artificial intelligence and biomechanics to make them as close to their human counterparts as possible in their locomotion, perceptual, cognitive and planning capabilities.

In this paper, we focus on the aspect of locomotion control for virtual characters so that they can not only find a way to reach their target position but do so in a natural and realistic manner. By employing a data-driven method, we can query from a set of animation clips and select the best one so that the character can walk, turn and stop given online input commands.

The system described above provides good results upto a user-defined degree of error without problems such as foot-skating. It can further to be used to explore more problems in virtual character animation by building functionalities on top of existing ones.

1 INTRODUCTION

Within the umbrella branch of computer animation, a lot of research work is being done to make virtual characters look and behave like their real life counterparts. Animation of human walking is a field of interest because of it's widespread applications in games, movies and various other fields. Much research is being done in the area of locomotion to render actions such as walking and running as realistically as possible because the keen human eye can easily identify between any artifact in motion and natural fluent motion.

Animation of virtual humans is a complex process which involves synthesizing the motion first for a *skeleton* after which this skeleton can be coated with deformable surfaces modeling skin or clothes [1]. A virtual human is represented as a hierarchical set of interconnected bones each with their own degrees of freedom stored in a state vector as shown in Figure 1. How this state vector changes over time describes the motion of the character.

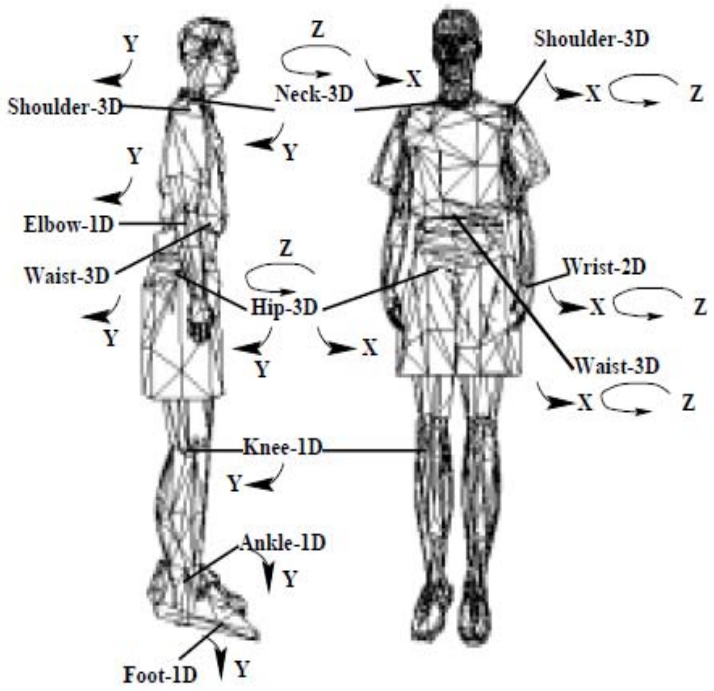


Figure 1 Articulated figure with thirty degrees of freedom

The main idea behind human walking (abstracted from [1]) is:

Researcher in biomechanics characterize human walking as the succession of phases separated by footstrikes FS (the foot is in contact with the ground) and takeoffs TO (the foot leaves the ground). In gait terminology, a stride is defined as a complete cycle from a left foot takeoff to another left foot takeoff, while the part of the cycle between the takeoff events occur during a stride: left takeoff (lTO), left footstrike (lFS), right takeoff

(rTO), and right footstrike (rFS). This leads to the characterization of two motion phases for each leg as shown in Figure 2:

1. The period of support which is referred to as the *stance phase,*
2. The period of non-support which is known as the *swing phase.*

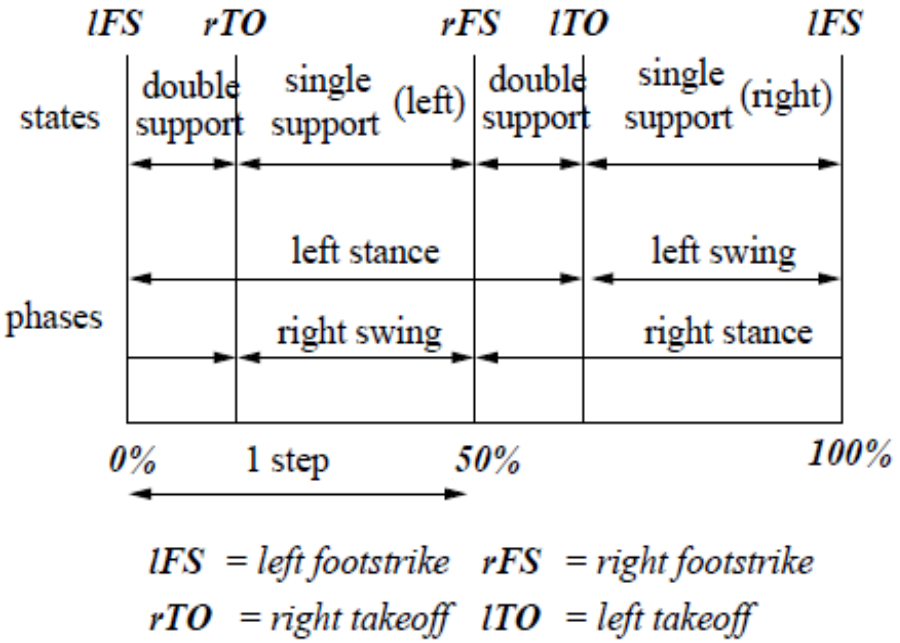


Figure 2 Characteristics phases of the walking motion

Virtual human locomotion can be broadly classified into three classes as described below.

1.1 Procedural Animation

The basic idea underlying procedural or kinematic animation is the use of *forward kinematics* and *inverse kinematics* to get updated state vector of the articulated figure at each time step and thus synthesize motion. Taking as input the change in end effector position ΔX, the change in joint angles $\Delta\theta$ can be calculated by the use of forward kinematics with the formula:

$$\Delta X = \mathrm{J}\Delta\theta$$

where J is the Jacobian matrix which is the matrix of all first-order partial derivatives of a vector function with respect to another vector.

By applying this formula to the key-frames created by the designer and using interpolation techniques to obtain in-between frames, motion can be computed. Using procedural animation techniques gives the user a high-level control such as configuring velocity and step-length but suffers from drawbacks such as decoupling of joints and foot penetration since a particular end effector position can be achieved by potentially various different configurations of the joints. These can be corrected using the technique of *inverse kinematics* by reversing the above formula which is beyond the scope of this paper.

1.2 Physics-based techniques

Physics-based techniques are used to provide realism to the motion in cases where the actor has to respond to external stimuli such as laws of physics, external forces or torques and to capture the essence of the character's interaction with the changing virtual environment. Based on Newton's law of physics and Lagrangian dynamics, they use similar principles as kinematic animation and generate motion by simulating from root link onwards given external factors by maintaining joint constraints. A virtual character is rendered more lifelike with the use of these techniques and can successfully adjust it's posture and subsequently motion when encountered with situations such as walking up an elevated slope or carrying a heavy load. These techniques tend to be computationally expensive and generally require trade-offs between realism and performance.

1.3 Example-based motion

Out of the three classes of locomotion, the most recent ones are the example-based approaches or *motion capture* approach. Techniques are in place to use magnetic and optical technologies to store the position and orientation of points located on human body and adapt captured trajectories to synthetic skeleton [2]. Since the motion clips are pre-generated, the volume of the database tends to grow very large and there is little interactive control in these techniques. Within the branch of example-based approaches, there are many sub-branches such as *motion blending* and *motion warping* but the most

relevant to our work is the combination of blending algorithms and the step-space approach [3].

The underlying idea behind the step-space approach is to generate a database of step space using information about step coordinates, their angular displacement with respect to the supporting foot, the swing time and the stance time. Once the database is generated, using weighted nearest neighbor approach, a set of steps are found corresponding to the query footsteps in the foot plan given. The steps are planted on the foot plan and rotational corrections are done to make them as close to desired footsteps as possible. Finally, temporal and spatial warping take care of any mismatch in speed while transitioning and produce a neat animation without problems of footskating.

2 RELATED WORK

The work of Wenjia Huang [3] in implementation of data-driven locomotion control in OgreMax has been pivotal to our work and it's motion database provides a framework for our application to be built upon. Figure 3 shows the system overview of the above described work.

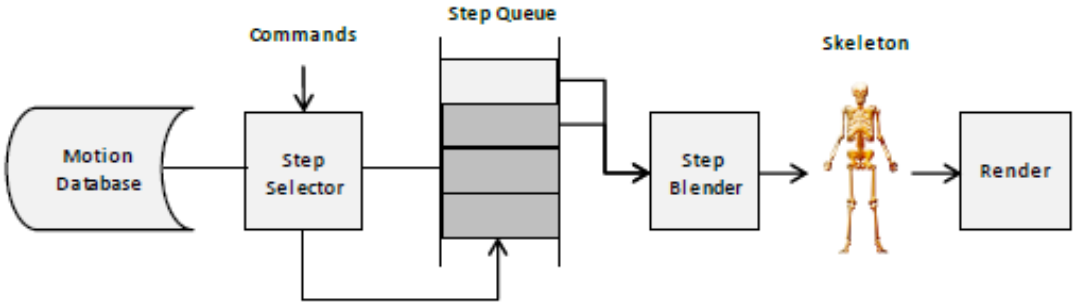


Figure 3 System overview and main components

The motion database referred to above contains animation clips of walking at different speeds and angles, and transition between walking and standing animation. It is used in the following way (abstracted from [3]):

The motion data is parametrized in step space, so that the step selector can access individual step clips and their corresponding step parameters. The step selector gets in the walking commands of type/speed/angle and also the stop facing angle, then it chooses the

best step clips from the motion database and pushes them in step queue. The step blender will access the first two steps in the queue, and ease-out the first step and ease-in the previous step of the second step, while the resulting data is applied to the skeleton of the character mesh rendered in 3D environment.

The character first turns towards the target position, and with the goal in mind, takes the appropriate steps. Towards the end of a particular animation clip, it queries the facing angle and required angle again and pushes steps in the queue to ease-in to. Using this technique, it can reach a distant location conveniently. The above system works well to achieve it's goals but will fall short in constrained environments due to the nature of the motion clips constructed in 3DS Max by placing footsteps and achieving the required turning angle.

The motion clips used in our work are derived from [3] as described in the table below.

Motion type	Description	Available step pattern(s)
Walk	Turn 180′ left	RLR
Walk	Turn 180′ right	LRL
Walk	Turn 135′ left	RLR
Walk	Turn 135′ right	LRL
Walk	Turn 90′ left	RLR
Walk	Turn 90′ right	LRL
Walk	Turn 45′ left	RLR
Walk	Turn 45′ right	LRL
Walk	Turn 10′ left	RLR
Walk	Turn 10′ right	LRL
Walk	Turn 20′ left	RLR
Walk	Turn 20′ right	LRL
Walk	Turn 30′ left	RLR
Walk	Turn 30′ right	LRL
Walk	Straight	LR RL
Walk to stand transition	Walk to stand	LR RL
Stand to walk transition	Stand to walk	LR RL
Walk to stand transition	Walk to stand 45′ left	RL
Walk to stand transition	Walk to stand 45′ right	LR
Walk to stand transition	Walk to stand 90′ left	RL
Walk to stand transition	Walk to stand 90′ right	LR
Walk	Straight slow	LR RL
Walk	Slow to normal	LR RL
Walk	Normal to slow	LR RL
Walk to stand transition	Slow walk to stand	RL LR
Walk to stand transition	Stand to slow walk	RL LR

Table 1 Motion Database

3 OUR WORK

As mentioned earlier, for our work we reuse motion clips from [3] and import them into Unity for our framework to be built upon. The following sections describe our work in detail.

3.1 Game engine selection

At the start of this work, the question we had was regarding which game engine suits our needs the best and should be picked. After looking at various engines such as Panda3D, Unity, Unreal Engine, we chose Unity due to the following reasons:

1. Unity is an academic standard while an engine like Unreal is industry standard. While there is an obvious performance trade-off in picking Unity, it's use is more practical to our situation and gives flexibility to add functionality to our modules later on if a researcher decides to pick it up since it's used widely in academic settings.
2. The learning curve for Unity is believed to be easier than that of other engines due to a wealth of available online documentation and it's easy to understand graphical user interface.
3. The file import feature for Unity is very simple and intuitive. It supports files from most modeling packages which can simply be dropped on to the working folder for import.

4. Unity supports many well-known scripting languages such as Javascript, C# and Boo whereas Unreal Engine exclusively uses UnrealScript and Panda3D uses Python only.

A game engine like Unreal surely provides more design tools and has a better physics engine for the purpose of our work, Unity meets all the requirements.

3.2 Motion clips preparation in 3DS Max and import in Unity

All the clips in our motion database have animation starting at the origin of the world coordinate system. Even though it's a good practice since it maintains consistency, it creates an issue as well. Imagine the following scenario:

Suppose there are two animation clips X and Y that need to be played back to back but only the second half of clip Y is required. In the present scenario, clip X is played at the end of which the character position is updated and the local coordinate system is set to zero. Unity provides the option of querying the length of the clip and playing an animation clip from a certain time onwards. On playing the latter half of clip Y, the problem that arises is that the character snaps to the position where it should have been had the whole clip been used and begins animation there instead of starting at it's immediate position. This breaks the smoothness in the motion and creates artifacts that look ugly.

Similar was the problem faced in our implementation during *motion blending* phase described later. We found an easy fix for it by modifying some motion clips in 3DS Max. Instead of now starting at the origin of world coordinate system, by using the *Move All Mode* under the *Biped* tab in *Motion*, we created an offset in the clips as shown in Figure 5 so that when they're played from a certain frame onwards, no artifacts are produced.

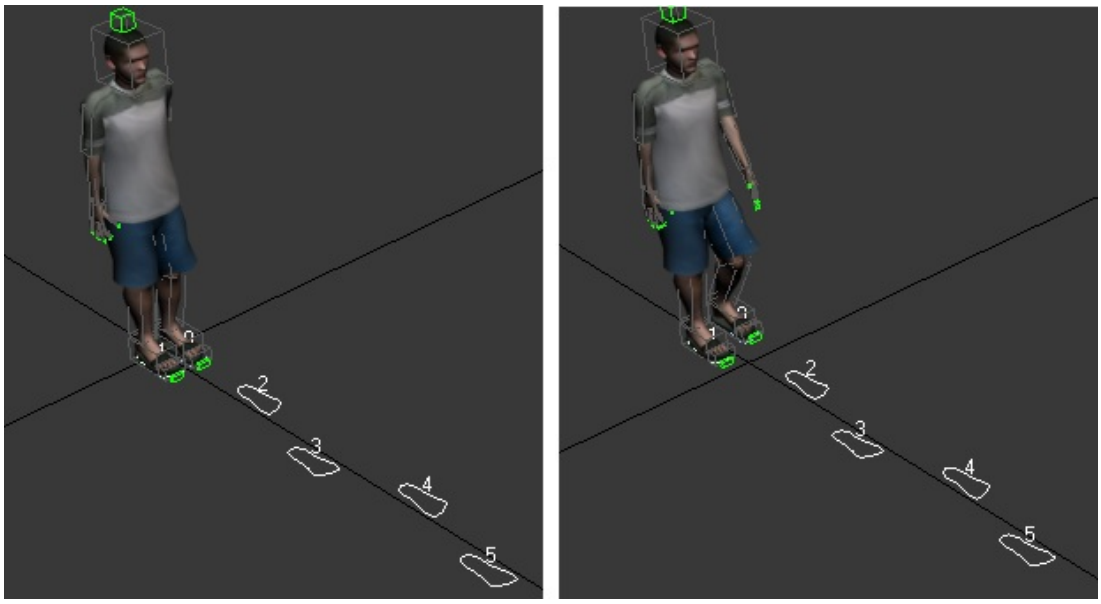


Figure 4 (a) Character at world origin (b) Character after offset

The next part involved uploading these *corrected* motion clips into Unity. Unity allows the user to import animations using the @ animation naming scheme. For different actions such as walking straight or turn 10 degrees left, one can just name the files like

Guy@walk_straight and Guy@turn_10_left. Upon importing these files, Unity collects all the animations under the header *Guy* and will set up to reference walk straight or turn 10 degrees left automatically. This is shown in figure 6 below. By dragging just the *Guy* asset onto the scene, one can now use all the animation clips attached to it.

Figure 5 Animation clips used for the character

3.3 Data-driven locomotion module

Our system consists of four scripts explained below:

1. Point to target: This script is used to display the target position for the character. It uses the Unity script function *Camera.ScreenPointToRay* which returns a ray going from the camera through a screen point. We find the world coordinates of the point where the ray intersects the plane and display a sphere as a target marker

there.

2. Camera target: This script simply describes where the camera should look at using the inbuilt function *transform.LookAt.*
3. Line renderer: This script is tied to the plane on which the character walks and draws a line between the position of the character at the start of the animation clip and the sphere marker at the target position.
4. Character controller: The character controller script controls which animation is played at any given point and is responsible for taking the character to the user-defined target position. It has a *start* function which sets all the parameters used in the script. In Unity, the primary function is the *update* function where all the game logic goes. It is called before rendering every frame.

The direction that the character is currently facing can be queried using the *transform.forward* function. Just for user reference, we draw a line projecting outwards from the character in the direction where it is looking to give user an idea of the angle between the current character position and the target position.

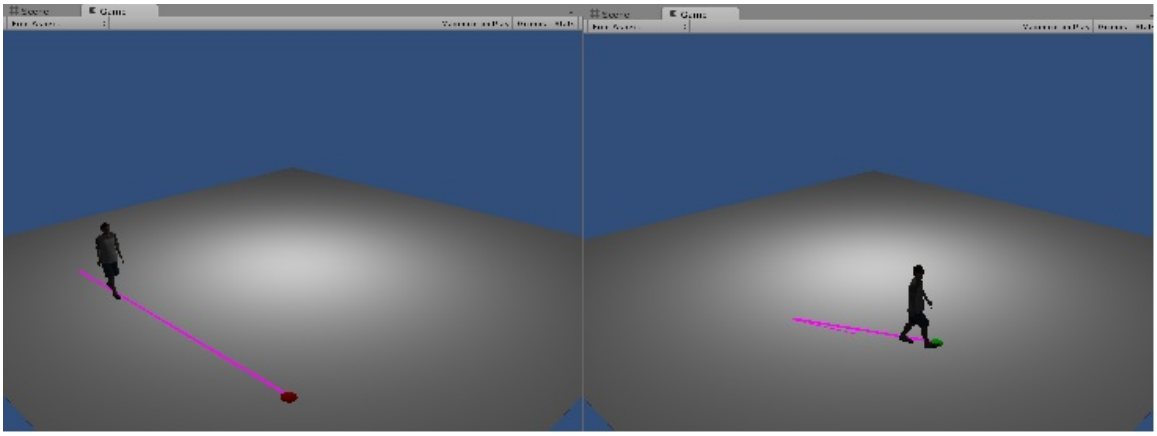
Next, we check whether the target is to the right or the left of the character simply by doing standard cross and dot product routines.

We then do a simple check to see if the character is already at the target position. If not, we play the appropriate clip. This is also the time we do the motion blending referred to in the earlier section. The way the motion clips are designed in 3DS Max, each animation begins with the character taking the left foot stride and each animation ends at a pose where the character has it's left knee bent. If two animation clips are played back to back, it creates artifacts in the motion because there is no proper transition between clips. To this effect, we have the *correction* clip. The *correction* clip blends motion by easing out the previous animation and easing in the next one. In essence, it is a one step animation which corrects the motion to make it smooth. Since the latter part of the *correction* clip is the same as the starting of every other motion clip, playing the *correction* clip upto a certain time, stopping it and starting the next one exactly where it is coincides with the *correction* clip gives good results. This careful manipulation prevents problems of footskating and produces a smooth motion without artifacts.

The only issue left then is to update the position and rotation of the character at the end of each animation clip. It is so because if it were not done, the character would snap back to the point where it started that particular animation.

4 RESULTS

The system produces good results. Given the target position online, the character is able to walk, turn and stop close to the destination in a satisfactory manner. For large turning angles, the character rotates on it's own axis thereby reducing the angle and then follows the footsteps corresponding to smaller angles. Figure 7 below shows snapshots of the



system.

Figure 6 (a) Character approaching it's target (b) Character at it's target position

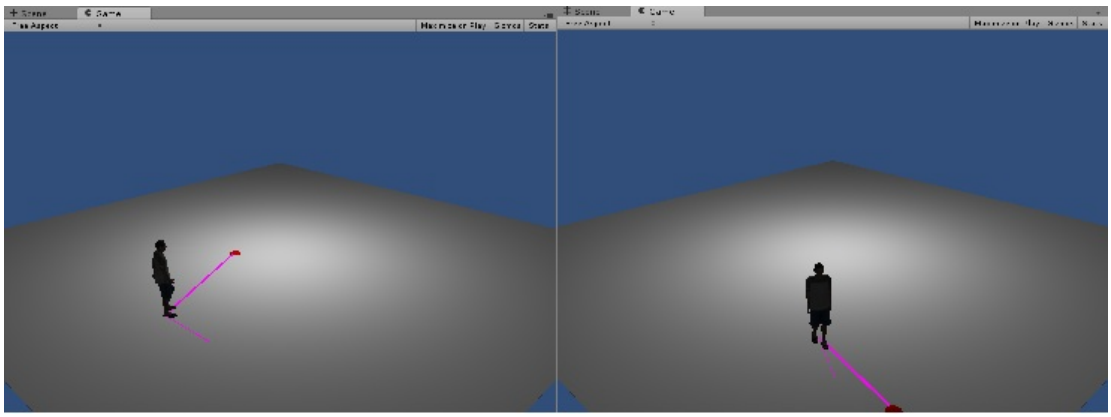


Figure 7 (a) Large turning angle (b) Small turning angle

5 APPENDIX – SOURCE CODE LISTING

1 Point to target

```
var plane : Plane = new Plane(Vector3.up, Vector3.zero);      //Defines a plane

function Start()
{       renderer.material.color = Color.red; }

/* This function converts the screen coordinates to world coordinates*/
function Update ()
{
        if(Input.GetMouseButton(0))                          //Check for mouse button down
        {
                /*Ray through mouse position*/
                var ray = Camera.main.ScreenPointToRay(Input.mousePosition);
                var ent : float = 100.0;
                if (plane.Raycast(ray, ent))
                {
                        var hitPoint : Vector3= ray.GetPoint(ent);
                        transform.position = hitPoint;
                }
        }}
```

2 Camera target

```
function Update ()
 {
        transform.LookAt(Vector3.zero);    //Look at world origin
}
```

3 Line renderer

```
function start()
{
        var lineRenderer1 : LineRenderer = gameObject.AddComponent(LineRenderer);
        lineRenderer1.SetWidth(0.5,0.5);
        lineRenderer1.SetVertexCount(2);
```

```
}
/* Draws a line from character to target marker*/
function Update ()
 {
        var lineRenderer1 : LineRenderer = GetComponent(LineRenderer);
        lineRenderer1.useWorldSpace = true;
        lineRenderer1.SetPosition(0,GameObject.Find("Guy").transform.position);
        lineRenderer1.SetPosition(1,GameObject.Find("Sphere").transform.position);
}
```

4 Character controller

```
private var myRigidbody : Rigidbody;
var a : int;
var b : Vector3;
var pos : Vector3;
var roty : Quaternion;
var newtarget : Vector3;
var resume_ok : int;
var check : int;

function Start()
{
        var linerenderer : LineRenderer = gameObject.AddComponent(LineRenderer);
        linerenderer.SetWidth(0.5,0.5);
        linerenderer.SetVertexCount(2);
        myRigidbody = rigidbody;
        a=1;   //variable to check whether character position has been updated

        pos=transform.Find("CMan0016-Bip").position;
        pos.y=8.7;

        check=0;  //variable that checks whether the correction clip has been played

        roty.x = transform.Find("CMan0016-Bip").localRotation.x;
        roty.y = transform.Find("CMan0016-Bip").localRotation.y;
        roty.z = transform.Find("CMan0016-Bip").localRotation.z;
        roty.w = transform.Find("CMan0016-Bip").localRotation.w;
```

```
	newtarget = GameObject.Find("Sphere").transform.position;

	resume_ok=1;   //Variable to make sure that the correction clip doesn't play in
loop
}

function Update ()
{
	var linerenderer : LineRenderer = GetComponent(LineRenderer);
	var temp : Vector3 =
Vector3((transform.position.x+(15*transform.forward.x)),(transform.position.y+(15*tran
sform.forward.y)),(transform.position.z+(15*transform.forward.z)));
	linerenderer.SetPosition(0,transform.position);
	linerenderer.SetPosition(1,temp);

	b=transform.position;
	var t1 = transform.forward;
	var t2 = GameObject.Find("Sphere").transform.position - transform.position;
	var angle = Vector3.Angle(t2,t1);
	var direct : int;
	var perp : Vector3 = Vector3.Cross(t1,t2);
	var dir : float = Vector3.Dot(perp,Vector3.up);
	/*Checks whether the target marker is to the left or right*/
	if (dir > 1.0)
	{
		direct = 1;
	}
	else if (dir < 1.0)
	{
		direct = -1;
	}
	else
	{
		direct = 0;
	}

	/*Checks if target has been updated*/
	if(newtarget!=GameObject.Find("Sphere").transform.position)
	{
```

```
		GameObject.Find("Sphere").renderer.material.color = Color.red;
		newtarget = GameObject.Find("Sphere").transform.position;
	}

	var sphere_color = GameObject.Find("Sphere").renderer.material.color;
	if(sphere_color==Color.red)
	{
	var distance_from_target : float =
(Mathf.Sqrt((Mathf.Pow(((transform.Find("CMan0016-Bip").position.x)-
(GameObject.Find("Sphere").transform.position.x)),2))+(Mathf.Pow(((transform.Find("C
Man0016-Bip").position.z)-(GameObject.Find("Sphere").transform.position.z)),2))));

	if(distance_from_target<=7.0)
	{
		GameObject.Find("Sphere").renderer.material.color = Color.green;
	}

/*Selects appropriate animation clip*/
	if(angle>=0 & ((angle<=67.5 & direct==-1) | (angle<=135 & direct==1)) &
resume_ok==1 & check==0 & a==1)
	{
		animation["Correction"].time=0.66;
		animation.Play("Correction");
		a=0;
		resume_ok=0;
		check=1;
	}

	if(animation["Correction"].time>1.1)
	{
		animation.Stop("Correction");
	}

	if(animation["walk_straight"].time>2.8)
	{
		animation.Stop("walk_straight");
	}

	if(animation["sharp_turn_45_left"].time>4.8)
```

```
{
	animation.Stop("sharp_turn_45_left");
}

if(animation["sharp_turn_45_right"].time>4.8)
{
	animation.Stop("sharp_turn_45_right");
}

if(animation["sharp_turn_90_right"].time>4.8)
{
	animation.Stop("sharp_turn_90_right");
}

if(animation["turn_10_left"].time>4.8)
{
	animation.Stop("turn_10_left");
}

if(animation["turn_20_left"].time>4.8)
{
	animation.Stop("turn_20_left");
}

if(animation["turn_30_left"].time>4.8)
{
	animation.Stop("turn_30_left");
}

if(animation["turn_10_right"].time>4.8)
{
	animation.Stop("turn_10_right");
}

if(animation["turn_20_right"].time>4.8)
{
	animation.Stop("turn_20_right");
}
```

```
if(animation["turn_30_right"].time>4.8)
{
	animation.Stop("turn_30_right");
}

if (angle>=0 & angle<=5 & a==1 & check==1)
{
	animation["walk_straight"].time=1.1;
	animation.Play ("walk_straight");
	a=0;
	check=0;
	resume_ok=1;
}

if(angle>5 & angle<=15 & a==1 & direct==1 & check==1)
{
	animation["turn_10_right"].time=1.1;
	animation.Play("turn_10_right");
	a=0;
	check=0;
	resume_ok=1;
}

if(angle>5 & angle<=15 & a==1 & direct==-1 & check==1)
{
	animation["turn_10_left"].time=1.1;
	animation.Play("turn_10_left");
	a=0;
	check=0;
	resume_ok=1;
}



if(angle>15 & angle<=25 & a==1 & direct==1 & check==1)
{
	animation["turn_20_right"].time=1.1;
	animation.Play("turn_20_right");
	resume_ok=1;
```

```
        a=0;
        check=0;
}

if(angle>15 & angle<=25 & a==1 & direct==-1 & check==1)
{
        animation["turn_20_left"].time=1.1;
        animation.Play("turn_20_left");
        resume_ok=1;
        a=0;
        check=0;
}

if(angle>25 & angle<=37.5 & a==1 & direct==1 & check==1)
{
        animation["turn_30_right"].time=1.1;
        animation.Play("turn_30_right");
        resume_ok=1;
        a=0;
        check=0;
}

if(angle>25 & angle<=37.5 & a==1 & direct==-1 & check==1)
{
        animation["turn_30_left"].time=1.1;
        animation.Play("turn_30_left");
        resume_ok=1;
        a=0;
        check=0;
}

if(angle>37.5 & angle<=67.5 & a==1 & direct==1 & check==1)
{
        animation["sharp_turn_45_right"].time=1.1;
        animation.Play("sharp_turn_45_right");
        resume_ok=1;
        a=0;
        check=0;
}
```

```
        if(angle>37.5 & angle<=67.5 & a==1 & direct==-1 & check==1)
        {
                animation["sharp_turn_45_left"].time=1.1;
                animation.Play("sharp_turn_45_left");
                resume_ok=1;
                a=0;
                check=0;
        }

        if(angle>67.5 & angle<=135 & a==1 & direct==1 & check==1)
        {
                animation["sharp_turn_90_right"].time=1.1;
                animation.Play("sharp_turn_90_right");
                resume_ok=1;
                a=0;
                check=0;
        }

        if(angle>67.5 & angle<=135 & a==1 & direct==-1)
        {
                animation.Play("stand_rotate_90_left");
                resume_ok=1;
                a=0;
                check=0;
        }

        if(angle>=135 & a==1)
        {
                animation.Play("stand_rotate_180_left");
                resume_ok=1;
                a=0;
                check=0;
        }

/*Updates character position at the end of each clip*/
        if(!animation.IsPlaying("Test") & !animation.IsPlaying("Correction") &
!animation.IsPlaying("walk_straight") & !animation.IsPlaying("turn_10_left") &
!animation.IsPlaying("turn_20_left") & !animation.IsPlaying("turn_30_left") &
```

```
!animation.IsPlaying("turn_10_right") & !animation.IsPlaying("turn_20_right") &
!animation.IsPlaying("turn_30_right") & !animation.IsPlaying("sharp_turn_45_right") &
!animation.IsPlaying("sharp_turn_45_left") &
!animation.IsPlaying("stand_rotate_90_left") &
!animation.IsPlaying("sharp_turn_90_right") &
!animation.IsPlaying("stand_rotate_180_left") & a==0)
        {
                var newpos : Vector3;
                newpos.x = transform.Find("CMan0016-Bip").position.x;
                newpos.y = 8.7;
                newpos.z = transform.Find("CMan0016-Bip").position.z;

                var newrot : Quaternion = transform.Find("CMan0016-
Bip").localRotation;

                transform.Find("CMan0016-Bip").localPosition = Vector3.zero;
                transform.Find("CMan0016-Bip").localPosition.y = 8.7;

                var rottemp : Quaternion;
                rottemp.x = roty.x;
                rottemp.y = roty.y;
                rottemp.z = roty.z;
                rottemp.w = roty.w;

                transform.Find("CMan0016-Bip").localRotation = Quaternion.identity;
                transform.Find("CMan0016-Bip").localRotation *=rottemp;
                transform.position += newpos - pos;
                transform.localRotation *= newrot * Quaternion.Inverse(roty);

                pos = transform.Find("CMan0016-Bip").position;
                pos.y = 8.7;
                roty = transform.Find("CMan0016-Bip").localRotation;

                a=1;
        }
        }
}
```

6 REFERENCES

[1] Franck Multon, Laure France, Marie-Paule Cani-Gascuel, Gilles Debunne. Computer Animation of Human Walking: a Survey. 1998.

[2] T. Molet, R. Boulic, and D. Thalmann. A real time anatomical converter for human motion capture. In *Eurographics workshop on Computer Animation and Simulation,* pages 79-94, September 1996.

[3] Wenjia Huang. www.cs.ucla.edu/~hwj/