# Kingdom Hearts Movement Tutorial

Made with 3D Game Studio (www.3dgamestudio.com)
Written by David Lancaster (www.rebelplanetcreations.com)

Model – Exile
Animations – Kotakide

A6 - 6.405 was used for this tutorial.

`Code to be added or replaced is blue`
`Old code, used as reference, is green`

# Table of Contents

# INTRODUCTION

Welcome to this tutorial. This is a tutorial which explains in detail how to code a working and effective movement script, similar to that of Kingdom Hearts, Zelda or Beyond Good and Evil. Along with providing step by step instructions on how to create such code, I will also be sharing many concepts and ideas I have discovered during my 5 years of experience with 3D Game Studio and c-script. This script covers smooth movement and combo combat. It introduces you to techniques I have learnt in coding movement. Movement code has endless possibilities and covering everything possible would take forever. I believe most to every thing that you see in a AAA title or console game movement wise can be created in 3DGS. Once you have hold of some tools to coding movement, you'll find that you can create anything that comes to mind, whether it be adding wall running and jumping like Prince of Persia, to rope balancing in Super Mario Sunshine, it's all possible, it just requires the experience and the work. I hope you find this tutorial a valuable tool which will help your game creation ventures.

**Prerequisites**

I am assuming you have a beginners or basic understanding of c-script, are able to control objects through c-script and have a general understanding of it's functions and language. If not, a lot of the code I will be explaining may not make much sense. You must also have an understanding of vectors, c-script vector functions, and basic trigonometry. You must know how to create a basic WED level, assign a script, add and assign an action to a model, add a block and texture it etc. Also understand the basics of MED.

This tutorial may not be as beginner friendly as I would have hoped, you will need a good understanding of c-script to understand the concepts presented here. There are so many things to understand just to grasp the simple concepts of movement that this may seem overwhelming.

Please note the following fat and narrow hull settings I recommend for this tutorial:

Fat: 64 48 8
Narrow: 32 32 0

**For game studio versions below 6.4**

The code in this tutorial uses "time" for movement calculations, in versions 6.4 or above "time_step" would be a better variable to use.

This tutorial uses c_move and c_scan.

For all gamestudio versions below 6.4 replace all occurrences of c_move found in "handle_movement" and "enemy_dummy" with the following:

```
move_mode = ignore_passable | glide;
ent_move(nullvector,my.move_x);
//c_move(my,nullvector,my.move_x,use_aabb | ignore_passable | glide);
```

Also change the occurrence of c_scan in "handle_movement" with the following:

```
vec_set(temp.x,vector(360,180,250));
result = scan_entity(player.x,temp);
//c_scan(player.x,player.pan,vector(360,180,250),scan_ents | scan_limit | ignore_me);
```

# MOVEMENT

## Basic Movement

Firstly, we will be using simple “key_” commands to determine which key is being pressed, I know there are ways of setting up key mapping which Orange_Brat has provided very good code for (below), but we wont be using key mapping in this tutorial:

http://www.coniserver.net/ubbthreads/showflat.php/Cat/0/Number/625816/an/0/page/0#Post625816 (key mapping)

Open WED and create a new level, save your level as “level.wmp”, add a new / blank script named main.wdl which contains the following code:

```
include <player.wdl>;

FUNCTION main(){
        level_load("level.wmb");
}
```

Now create a new .wdl file named player.wdl for now simply add this code in player.wdl:

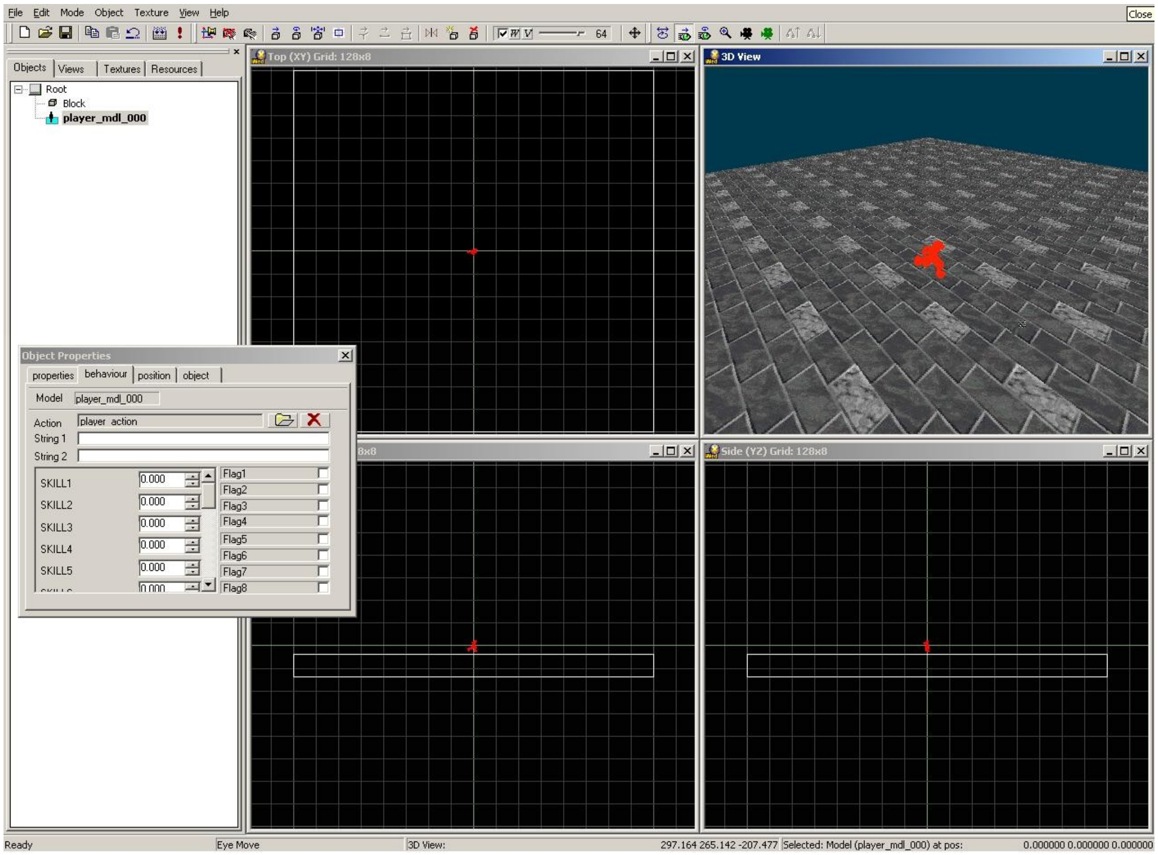
```
ACTION player_action {
        wait(1);
}
```

We will no longer be adding code to main.wdl except new script additions.

I like to work with a higher resolution so I will be adding this line to the main function:

```
video_switch(8,0,0);
```

Click map properties and change the nexus to 200.  Add a block to your level, a big long plane, then load the player.mdl entity that came with this tutorial and assign it the action “player_action” (please note that you may need to reset WED before this action appears in the list of actions to choose from).



Model player_mdl_000
Action player action
String 1
String 2
SKILL1 0.000
SKILL2 0.000
SKILL3 0.000
SKILL4 0.000
SKILL5 0.000
Flag1 ☐
Flag2 ☐
Flag3 ☐
Flag4 ☐
Flag5 ☐
Flag6 ☐
Flag7 ☐
Flag8 ☐

297.164 265.142 -207.477 Selected: Model (player_mdl_000) at pos: 0.000000 0.000000 0.000000

Now it's time to write the code for our player. We will start where all movement scripts start, with simplicity, but things may get more complex later on.

Open player.wdl. We are going to add some defines at the beginning of the script:

```
DEFINE move_x,skill22; //movement skills
DEFINE move_y,skill23;
DEFINE move_z,skill24;
DEFINE force_x,skill25;
DEFINE force_y,skill26;
DEFINE force_z,skill27;
DEFINE velocity_x,skill28;
DEFINE velocity_y,skill29;
DEFINE velocity_z,skill30;

DEFINE animate,SKILL31; //animation skills
DEFINE animate2,SKILL32;
DEFINE animblend,SKILL33;
DEFINE currentframe,SKILL34;
DEFINE blendframe,SKILL35;
DEFINE z_offset,SKILL50; //gravity and jumping skills
DEFINE jumping_mode,SKILL51;
DEFINE gravity,SKILL52;
DEFINE movement_mode,SKILL53; //for various movement mainly combat
DEFINE moving,SKILL54;

DEFINE hit_by_player,SKILL55; //enemy skills
DEFINE entity_type,SKILL56;
```

These defines are supplying a secondary name for skills, so now when we use "my.move_x" it is the same as typing "my.skill22". Also, c-script vector functions are very handy for example, if we enter a skill into a vector function "vec_set(my.skill22,temp);" the script will see my.skill22 as the x component of the vector and automatically look at the next 2 skills, skill23 and skill24 for the y and z component. So if we put my.move_x into a vector function, my.move_y (my.skill23) and my.move_z (my.skill24) will be used as the y and z components.

Now replace the current "player_action" with the following code:

```
ACTION player_action {
        my.shadow = on;
        WHILE (1) { //the main loop
                handle_movement();
                handle_camera();
                wait(1);
        }
}

FUNCTION handle_movement() {
        temp.x = -1000;
        temp.y = 0;
        IF (key_w == 1) { temp.x = camera.pan; }
        IF (key_s == 1) { temp.x = camera.pan + 180; }
        IF (key_a == 1) { temp.x = camera.pan + 90; }
        IF (key_d == 1) { temp.x = camera.pan - 90; }
        IF (temp.x != -1000) { temp.y = 15 * time; }
        my.move_x = fcos(temp.x,temp.y);
        my.move_y = fsin(temp.x,temp.y);
        c_move(my,nullvector,my.move_x,use_aabb | ignore_passable | glide);
}

FUNCTION handle_camera() {
        vec_set(camera.x,vector(my.x + fcos(my.pan,-200),my.y + fsin(my.pan,-200),my.z + 
80)); //place the camera behind the player
        vec_diff(temp.x,my.x,camera.x); //make the camera look towards the player
        vec_to_angle(camera.pan,temp.x);
}
```

This may seem easily understood or very very confusing, either way I'll go through and explain each line of this code:

```
ACTION player_action {
      my.shadow = on;
      WHILE (1) { //the main loop
            handle_movement();
            handle_camera();
            wait(1);
      }
}
```

This is the player's action, 2 functions are being called every frame "handle_movement" and "handle_camera", we put such code in functions for simplicity, to separate the camera from the movement etc, this would work just the same if you replaced the "handle_movement()" line with the actual code in the function.

```
FUNCTION handle_movement() {
      temp.x = -1000;
      temp.y = 0;
      IF (key_w == 1) { temp.x = camera.pan; }
      IF (key_s == 1) { temp.x = camera.pan + 180; }
      IF (key_a == 1) { temp.x = camera.pan + 90; }
      IF (key_d == 1) { temp.x = camera.pan - 90; }
      IF (temp.x != -1000) { temp.y = 15 * time; }
      my.move_x = fcos(temp.x,temp.y);
      my.move_y = fsin(temp.x,temp.y);
      c_move(my,nullvector,my.move_x,use_aabb | ignore_passable | glide);
}
```

Temp.x and temp.y are variables that we will use for many purposes, consider it a temporary variable.  I assume you understand c_move, and the difference between relative and absolute movement.  In the above code temp.y is the force the player moves and temp.x is the direction (angle).  Temp is a predefined engine vector, that can be used as a temporary variable.

Temp.y is the movement speed, temp.x is initially set to -1000, if the player is holding WSAD temp.x will be set to a value between -135 and 540 and as such temp.y is set to "15 * time" rather than 0.   "IF (temp.x != -1000) { temp.y = 15 * time; }"

Temp.y needs to be reset to 0 each frame otherwise if you let go of WSAD the player would still move, as temp.y was left at it's last value, also the temp.y value is changed by other functions so the movement speed could potentially be messy if we don't reset it.

Temp.x is the angle the player moves, if we are holding W we want the player to move in the direction of the camera (or the way the camera is facing) "temp.x = camera.pan;", if we are holding A we want the player to move in a positive 90 degree direction from where the camera is facing (ie to the left) etc etc.

I am also assuming you understand basic trigonometry and the purpose of the fcos and fsin functions.

My.move_x and my.move_y (the x and y movement components used in c_move) are being set to the direction the player needs to move based on which key is being pressed.

```
FUNCTION handle_camera() {
      vec_set(camera.x,vector(my.x + fcos(my.pan,-200),my.y + fsin(my.pan,-200),my.z + 
80));
      vec_diff(temp.x,my.x,camera.x);
      vec_to_angle(camera.pan,temp.x);
}
```

This function is once again using trigonometry to place the camera behind the player based on the player's angle (the player moves relative to the camera's pan, the camera is placed based on the player's pan).

```
vec_set(camera.x,vector(my.x + fcos(my.pan,-200),my.y + fsin(my.pan,-200),my.z + 80));
```

This is a shorter and faster version of typing the following:

```
camera.x = my.x + fcos(my.pan,-200);
camera.y = my.y + fsin(my.pan,-200);
camera.z = my.z + 80;
```

The camera is placed at the player's position and moved back 200 quants behind it, the camera is placed at a z position of 80 quants above the player's z coordinate.

```
vec_diff(temp.x,my.x,camera.x);
vec_to_angle(camera.pan,temp.x);
```

Vec_diff finds the vector from the camera to the player, vec_to_angle gets the angle of this vector and assigns it to the camera, ie to make the camera look towards the player.  We will look into more advanced camera collision later.

Build and run your level, some basic, not very good looking, movement should be in place.  Yay!  But our character doesn't move diagonally, so let's do that:

Find this code:

```
IF (key_w == 1) { temp.x = camera.pan; }
IF (key_s == 1) { temp.x = camera.pan + 180; }
IF (key_a == 1) { temp.x = camera.pan + 90; }
IF (key_d == 1) { temp.x = camera.pan - 90; }
```

And replace with the following:

```
IF (key_w == 1 && key_s == 0 && key_a == 0 && key_d == 0) { temp.x = camera.pan; }
IF (key_s == 1 && key_w == 0 && key_a == 0 && key_d == 0) { temp.x = camera.pan + 180; }
IF (key_a == 1 && key_s == 0 && key_w == 0 && key_d == 0) { temp.x = camera.pan + 90; }
IF (key_d == 1 && key_s == 0 && key_a == 0 && key_w == 0) { temp.x = camera.pan - 90; }
IF (key_w == 1 && key_a == 1 && key_d == 0 && key_s == 0) { temp.x = camera.pan + 45; }
IF (key_w == 1 && key_d == 1 && key_a == 0 && key_s == 0) { temp.x = camera.pan - 45; }
IF (key_s == 1 && key_a == 1 && key_d == 0 && key_w == 0) { temp.x = camera.pan + 135; }
IF (key_s == 1 && key_d == 1 && key_a == 0 && key_w == 0) { temp.x = camera.pan - 135; }
```

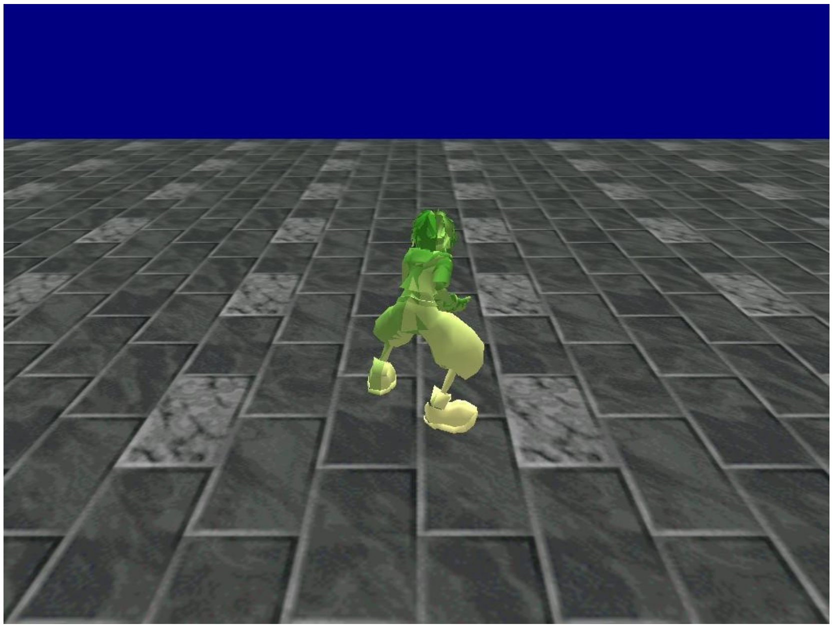
This is getting a little more controlled in regards to our controls.  Now the player will only move forward if W is behind held down and ASD are not.  The first 4 lines do the same as before but make sure the other keys aren't being held down.  The last 4 lines move the player diagonally if we are holding the corresponding keys, ie if WA are being held move the player in a positive 45 degrees from the camera's pan, ie to the upper left.

Now our player moves, kinda well, but there's no animations or gravity! And the camera doesn't even turn. Let's start with animations in the next section.

## Animations

If you don't understand how to animate an entity, now would be a good time to check the 3DGS manual regarding ent_animate. I am assuming you know how to use ent_animate to control an entities animations. I am also going to assume you know how to use ent_blend, I understand that can be a tricky function to get the grasp of and will explain a little below.

Setting up good animation framework is crucial for simplicity in controlling animations.



Firstly let's add a new script, make the following changes to main.wdl:

```
include <player.wdl>;
include <animation.wdl>;
```

Create a new wdl file named animation.wdl. We will be adding the "handle_animation" function here, please note that whenever I refer to "handle_animation" function in this tutorial the changes will need to be made in the animation.wdl file.

Using our 5 animation skills I will show you how to create simple framework, where you only need to call 1 function to control an entities animations and change 1 variable to change it's animations. Animating an entity is simple, setting up the animations to blend from one to another, and so the blend process isn't interrupted until it is fully completed makes coding animations a little more difficult.

```
DEFINE animate,SKILL31; //our 5 animation skills (this code has already been added)
DEFINE animate2,SKILL32;
DEFINE animblend,SKILL33;
DEFINE currentframe,SKILL34;
DEFINE blendframe,SKILL35;
```

Okay let's start by creating a function below all of our movement code in player.wdl, (alternatively you could create an animation.wdl file and place the code separately in there, make sure that the animation.wdl file is included before player.wdl so that player.wdl an recognize it's functions).

```
FUNCTION handle_animation(animation_speed) {
      IF (animation_speed <= 0) { animation_speed = 1; }
      IF (my.animblend != blend && my.blendframe != nullframe) { my.animate2 = 0;
my.animblend = blend; }
      IF (my.animblend == blend) {
            IF (my.currentframe == stand)
{ ent_animate(my,"stand",my.animate,anm_cycle); }
            IF (my.currentframe == run) { ent_animate(my,"run",my.animate,anm_cycle); }
            IF (my.currentframe == walk) { ent_animate(my,"walk",my.animate,anm_cycle); }
            IF (my.blendframe == stand) { ent_blend("stand",0,my.animate2); }
            IF (my.blendframe == run) { ent_blend("run",0,my.animate2); }
            IF (my.blendframe == walk) { ent_blend("walk",0,my.animate2); }
            my.animate2 += 45 * time;
            IF (my.animate2 >= 100) {
                  my.animate = 0;
                  my.animblend = my.blendframe;
                  my.blendframe = nullframe;
            }
      }
      IF (my.animblend == stand) {
            ent_animate(my,"stand",my.animate,anm_cycle);
            my.animate += 5 * animation_speed * time;
            my.animate %= 100;
            my.currentframe = stand;
      }
      IF (my.animblend == run) {
            ent_animate(my,"run",my.animate,anm_cycle);
            my.animate += 6.8 * animation_speed * time;
            my.animate %= 100;
            my.currentframe = run;
      }
      IF (my.animblend == walk) {
            ent_animate(my,"walk",my.animate,anm_cycle);
            my.animate += 8 * animation_speed * time;
            my.animate %= 100;
            my.currentframe = walk;
      }
}
```

I understand this function may be very very daunting, at the sheer perhaps complexity of it. I will attempt to explain how it works as best I can.

Next we are going to create some defines to make our animation coding easier, place this at the top of player.wdl with the skill defines, whenever this tutorial asks you to add a new define or variable, it will always be at the top of player.wdl and no other script:

```
DEFINE nullframe,-2;
DEFINE blend,-1;
DEFINE stand,0;
DEFINE run,1;
DEFINE walk,2;
DEFINE jump,3;
DEFINE fall,4;
```

Now whenever we refer to “blend” in a calculation or if statement etc, script will refer to it as the number -1, “stand” as 0 etc the following run + walk, would equal 3. Also keep in mind with our defines that something like “IF (my.animblend == stand)” would be the same as typing “IF (my.skill33 == 0)”, using defines allows us to use words which makes coding easier.

To begin with, we will simply use stand, walk and run animations, as we progress into combat and jumping we will use more.

In the “player_action” main loop add the handle_animation function:

```
handle_camera();
handle_animation(1);
```

Now go into the “handle_movement” function and add the following code after our c_move instruction:

```
c_move(my,nullvector,my.move_x,use_aabb | ignore_passable | glide);

IF (my.move_x != 0 || my.move_y != 0) { //if we are moving
      IF (my.animblend == stand) { //if our current animation is stand
            IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe = run; }
      }
      IF (my.animblend == run && key_shift == 1) { my.blendframe = walk; }
      IF (my.animblend == walk && key_shift == 0) { my.blendframe = run; }
} ELSE {
      IF (my.animblend > stand) { //if we aren't moving and our current animation is walk
or run, blend and cycle the stand animation
            my.blendframe = stand;
      }
}
```

Now run your level, hey your player is animated! If you hold shift he walks, let go he runs, stop running he's idle, and all the animations are blended too! Time to explain how this is done.

The to change our player's animation is to simply change one variable, my.blendframe. If we set my.blendframe to walk, the animation will blend from the current animation to the walk frame then cycle the walk frame. Very handy. my.animblend is the current action the animation is undertaking, if my.animblend == walk then the player is playing the walk animation, if it is stand the player is playing the stand animation. If my.animblend == blend then the player is blending from the current animation to the animation specified in my.blendframe. Let's take a closer look at the “handle_animation” function:

```
      IF (animation_speed <= 0) { animation_speed = 1; }
```

This line simply sets the local variable animation_speed to 1 if it is less than or equal to 0, this is to stop the animation freezing or playing backwards in case we somehow call the handle_animation function and don't specify an animation speed, this can/may be removed or changed to suit the sort of animation frame work you wish to set up. Animation speed is a local variable which we wont have much use for in this tutorial, but is a good thing to have if this function is used by more than one entity which requires animations to be played at different speeds.

```
      IF (my.animblend != blend && my.blendframe != nullframe) { my.animate2 = 0;
my.animblend = blend; }
```

This line of code is what begins the blending of an animation, the first part, my.animblend != blend, is checking to see if the animation is already blending or not, the second part, my.blendframe != nullframe, is making sure that there has been an animation change through code, ie my.blendframe = walk.  Notice that my.blendframe is set to nullframe at the end of the blending process, that way the blending process cannot begin until we have have given a new animation to change to.  my.animblend = blend is setting the animation mode to blend. my.animate2 is the variable used in the blending process, at 0 the animation blending is 0% interpolated, at 50 the animation is 50% interpolated, at 100 it is complete.

```
      IF (my.animblend == blend) {
            IF (my.currentframe == stand)
{ ent_animate(my,"stand",my.animate,anm_cycle); }
            IF (my.currentframe == run) { ent_animate(my,"run",my.animate,anm_cycle); }
            IF (my.currentframe == walk) { ent_animate(my,"walk",my.animate,anm_cycle); }
            IF (my.blendframe == stand) { ent_blend("stand",0,my.animate2); }
            IF (my.blendframe == run) { ent_blend("run",0,my.animate2); }
            IF (my.blendframe == walk) { ent_blend("walk",0,my.animate2); }
            my.animate2 += 45 * time;
            IF (my.animate2 >= 100) {
                  my.animate = 0;
                  my.animblend = my.blendframe;
                  my.blendframe = nullframe;
            }
      }
```

This is the blending process.  The last animation the player was using is stored in the my.currentframe variable, so if we were standing “my.currentframe == stand” set the animation to the last position the stand animation was at, notice that the my.animate variable doesn't change in the blending process, it stores the position the last animation was at. ent_animate sets the animation, ent_blend blends/interpolates from that animation to a new animation at a % rate, notice that each frame we need to reset the original animation through ent_animate and then ent_blend an increased % than the previous frame.  The rate of increase is in the my.animate2 += 45 * time; instruction.  We are blending to the beginning of the my.blendframe animation, that is why we have a 0 in the ent_blend instruction, if you wanted to blend to middle of the animation you would have 50 instead, for the end of the animation have 100 etc.

Once the blend process is completed “IF (my.animate2 >= 100)” the current animation “my.animblend” is set to the animation that we just blended to “my.blendframe”, my.animate is set to 0 so we can continue to animate that animation from the beginning of it.  my.blendframe is set to nullframe so no blending will occur again until my.blendframe is changed to the name of the new animation to blend to.

```
      IF (my.animblend == stand) {
            ent_animate(my,"stand",my.animate,anm_cycle);
            my.animate += 5 * animation_speed * time;
            my.animate %= 100;
            my.currentframe = stand;
      }
```

I hope this code speaks for itself, if my.animblend is equal to stand, play the stand animation.  “my.animate %= 100;” is the same as typing “IF (my.animate >= 100) { my.animate -= 100;}” but it is faster and less code.

my.currentframe is set to that animation, so that when we blend we know what the last animation being played was.

So I hope this explanation hasn't gone over your head, if it has don't worry, continue with this tutorial and learn as much as you can, even if it is very confusing. The more you learn and the more experience you get, you will be able to understand much more complex things, and eventually be able to come back to something like this and understand it instantly.

Now how are we controlling these animations? Remember in our player's main loop we are calling “handle_animation”, but how are we changing what animation to play? We are doing that in the “handle_movement” function, after the c_move instruction. Let's look at how we are managing this:

```
IF (my.move_x != 0 || my.move_y != 0) {
```

Firstly we determine if we are moving or not, this is necessary so we know whether to play a moving animation like walk or run, or to play the stand animation.

```
IF (my.animblend == stand) { //if our current animation is stand
      IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe = run; }
}
```

If we are moving, and our current animation is stand “IF (my.animblend == stand)” then we want to play the walk or run animation, we play the walk animation if the player is holding shift, the run if not. Remember the moment we set “my.blendframe = walk;” etc my.animblend is immediately set to the blend value at the beginning of the “handle_animation” function, so the if statement “IF (my.animblend == stand)” will no longer be true.

```
IF (my.animblend == run && key_shift == 1) { my.blendframe = walk; }
IF (my.animblend == walk && key_shift == 0) { my.blendframe = run; }
```

If the player is already playing the run animation and we are holding shift, change the animation to walk.
If the player is already playing the walk animation and we are not holding shift, change the animation to run.
This allows us to switch between the run and walk animations whilst moving.

```
IF (my.animblend > stand) {
      my.blendframe = stand;
}
```

This code is in the else section to the “IF (my.move_x != 0 || my.move_y != 0)“ part, this code is only looked at if the player is not moving. If our current animation is > than stand, which is really saying if our current animation is either walk or run, as stand = 0 , walk = 1 and run = 2 (we set this in our defines). Then we are going to change our animation to stand.

Congratulations, we just finished animations, I hope that made sense. Let's move on to rotating the player and camera.

## Rotation

Add the following function to player.wdl:

```
FUNCTION rotate_entity(rotate_angle,rotate_speed) {
      IF (my.pan == rotate_angle) { return; }
      result = ang(rotate_angle - my.pan);
      IF (result > 0) { my.pan += rotate_speed * time; }
      IF (result < 0) { my.pan -= rotate_speed * time; }
      IF (ang(rotate_angle - my.pan) < 0 && result > 0) { my.pan = rotate_angle; }
      IF (ang(rotate_angle - my.pan) > 0 && result < 0) { my.pan = rotate_angle; }
}
```

At the very top of player.wdl add this line:

```
FUNCTION rotate_entity(rotate_angle,rotate_speed);
```

We need to add this line because the function contains more than one local variable and if the function is defined below other functions. This simply has to do with the order of functions in c-script. If you have the function rotate_entity below another function that calls it you will get an error that the script cannot find the function, you either have to move the function above the function calling it or place a line like the one above above the functions that use it.

Make the following changes to “handle_movement”

```
c_move(my,nullvector,my.move_x,use_aabb | ignore_passable | glide);
IF (temp.y > 0) { rotate_entity(temp.x,20); }
IF (my.move_x != 0 || my.move_y != 0) { //if we are moving
```

If temp.y > 0 that means the player is moving, if we are moving rotate the player towards the angle stored in temp.x at a speed of 20. We don't want the player rotating if we aren't moving.

Replace:

```
IF (temp.x != -1000) { temp.y = 15 * time; }
```

```
IF (temp.x != -1000) {
      IF (key_shift == 1) { temp.y = 10 * time; } ELSE { temp.y = 15 * time; }
}
```

Now when we walk by holding shift, we will move slower than if we are running.

Add the following variable above the “player_action” action:

```
var camera_distance = 200; //distance camera is from player
```

Camera Rotation is a three array variable, camera_rotation.x will be used for the position the camera is around the player, camera_rotation.z will be used for the position of the camera.

Replace the “handle_camera” function with the following code:

```
FUNCTION handle_camera() {
      camera.pan -= mouse_force.x * 12 * time;
      camera.tilt += mouse_force.y * 8 * time;
      camera.tilt = clamp(camera.tilt,-30,10);
      temp = fcos(camera.tilt,-camera_distance);
      vec_set(camera.x,vector(my.x + fcos(camera.pan,temp),my.y +
fsin(camera.pan,temp),my.z + 20 + fsin(camera.tilt,-camera_distance)));
}
```

Run your level, move the mouse to rotate the camera, move the player around and watch it rotate.

`Let's look at the “rotate_entity” function.`

This is a very useful function, it can be used by any entity. Please note that I often use temp and result etc to store temporary values, these variables may be changed by many functions (even 3DGS ones) so you have to be careful that the value in these variables do not change between it's area of use otherwise you can get weird results. You may want to use skills or global/local variables.

```
IF (my.pan == rotate_angle) { return; }
```

If our current pan is equal to the angle we wish to rotate to leave the function early.

```
result = ang(rotate_angle – my.pan);
```

Find the angle between our current angle and the angle we wish to rotate to. This will return a value between -180 and 180. Zero being our pan relative to rotate_angle, result being the difference between the angles.

```
IF (result > 0) { my.pan += rotate_speed * time; }
IF (result < 0) { my.pan -= rotate_speed * time; }
```

If the angle we need to rotate to is higher than our angle we rotate a positive force. If that angle is lower we rotate a negative force.

```
IF (ang(rotate_angle - my.pan) < 0 && result > 0) { my.pan = rotate_angle; }
IF (ang(rotate_angle - my.pan) > 0 && result < 0) { my.pan = rotate_angle; }
```

If the player is panning in a positive direction, then suddenly passes the angle it was suppose to stop at, normally the player would begin panning in a negative direction the next frame, this results in a bobbing type movement. The first if statement is checking to see “If we are moving in a positive direction (result > 0) and if our angle has passed it's destination (ang(rotate_angle - my.pan) < 0) then stop panning, we set the player's pan to the rotate_angle value.

Time to look at our camera function:

```
camera.pan -= mouse_force.x * 12 * time;
camera.tilt += mouse_force.y * 8 * time;
camera.tilt = clamp(camera.tilt,-30,10);
```

Okay this may not be the best way to do it, but it is a way :) As we move the mouse left and right camera.pan changes. As we move the mouse up and down camera.tilt changes. Camera.tilt is being clamped between -30 and 10, the clamp instruction stops camera.tilt from going lower than -30 and higher than 10.

```
temp = fcos(camera.tilt,-camera_distance);
vec_set(camera.x,vector(my.x + fcos(camera.pan,temp),my.y + fsin(camera.pan,temp),my.z +
20 + fsin(camera.tilt,-camera_distance)));
```

This part may be a little tricky to explain, it involves some trigonometry. Temp value is being stored based on the camera's tilt, this way if the camera is up high tilting down low on the player, the camera will be a little closer on it's x/y plane than if the tilt were at 0. So imagine a sphere being around the player and the camera can only be at any point on this sphere. If you move the camera upwards (z wise), the camera is going to move in closer to the center x and y wise. This may not be needed but it's a cool thing to know. The other option is just to move the camera away from the player xy wise based on camera_distance, and keep the z movement separate from that. The instruction is similar to our first except that now the z distance the camera is from the player is based on the camera's tilt. my.z + 20 is included to place the camera 20 quants above where it'd normally be, this sort of values you'd sort around with to get the camera in the best position to see your player from.

Okay we've got camera movement and rotating completed. Next is gravity, we'll move onto camera collision later :)

## Gravity

There are a lot of ways to attempt gravity, the ellipsoid hull system makes it easier as the hull doesn't tend to fall through the ground, but I've come across issues with stepping over ledges and the player hovering in mid air as it's waist is holding it in mid air, if anyone has a solution to this please let me know :)

We will be using normal trace and not c_trace, I attempted it and couldn't get it to work, if anyone knows how to get c_trace to work for this code please let me know how :D

Add the following function to player.wdl:

```
FUNCTION handle_gravity() {
	trace_mode = ignore_me+ignore_passable+use_box;
	result = trace(vector(my.x,my.y,my.z - my.z_offset),vector(my.x,my.y,my.z - 4000));

	IF (result < 3) {
		my.force_z = -1 * result;
	} ELSE {
		my.force_z -= 6 * time;
		my.force_z = max(-30,my.force_z);
	}
	my.velocity_z += (time * my.force_z) - (min(time*0.7,1) * my.velocity_z);
	my.move_z = my.velocity_z * time;
}
```

Add the following to the player's action:

```
ACTION player_action {
	my.z_offset = 6;
	my.shadow = on;
	WHILE (1) { //the main loop
		handle_gravity();
		handle_movement();
```

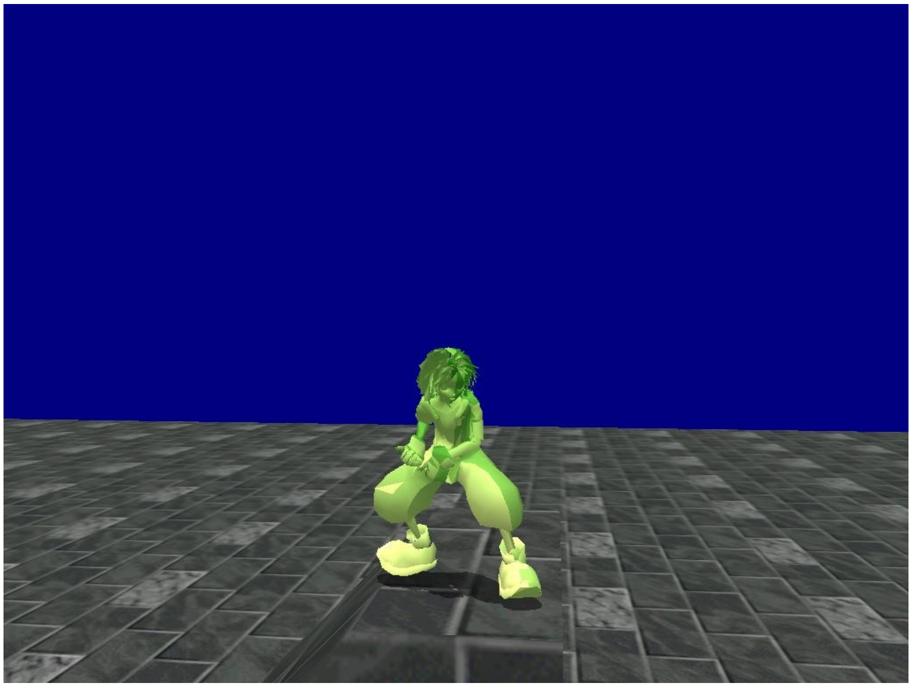
Add the following to the “handle_movement” function:

```
c_move(my,nullvector,my.move_x,use_aabb | ignore_passable | glide);

result = trace(vector(my.x,my.y,my.z - my.z_offset),vector(my.x,my.y,my.z - 4000));
IF (result < 0) { my.z -= result; my.velocity_z = 0; }

IF (temp.y > 0) { rotate_entity(temp.x,20); }
```

Now run your level, add a slope or 2, go climbing and falling, we got gravity working!



Time to explain the code. Let's start with the "handle_gravity" function:

```
trace_mode = ignore_me+ignore_passable+use_box;
result = trace(vector(my.x,my.y,my.z - my.z_offset),vector(my.x,my.y,my.z - 4000));
```

We are tracing from the player to 4000 quants below the player, we are using use_box to use the entities' hull for tracing, this doesn't always work correctly with AABB collision in regards to getting the player's feet to touch the ground, so that's where my.z_offset comes in, it is a value you enter in for a specific entity and it will offset where the gravity trace begins and results in how far the player is offsetted from the ground. For the model we are using we've set this value at 6 "my.z_offset = 6;"

```
IF (result < 3) {
      my.force_z = -1 * result;
} ELSE {
      my.force_z -= 6 * time;
      my.force_z = max(-30,my.force_z);
}
```

Result is the distance from the player's feet to the ground, if it is below 0 then it means the trace was inside a block and result returns a negative value, which means the player must move upwards rather than down, see trace in the gamestudio manual for more information. If the player is within 3 quants of the ground make the force which pulls the player to the ground based on the distance the player is to the ground, more so for a smooth/soft landing. If the player is above the ground (the else part) then increase the z_force and make sure it doesn't go below -30.

```
my.velocity_z += (time * my.force_z) - (min(time*0.7,1) * my.velocity_z);
my.move_z = my.velocity_z * time;
```

Accelerate the variable my.velocity_z based on my.force_z.. This is similar to the accelerate function (poor man's physics), feel free to try it, I put this equation in the script instead of accelerate as it was behaving in a few ways I didn't like. my.move_z is then calculated and used in the c_move instruction. That's our gravity.

```
result = trace(vector(my.x,my.y,my.z - my.z_offset),vector(my.x,my.y,my.z - 4000));
IF (result < 0) { my.z -= result; my.velocity_z = 0; }
```

After performing c_move, sometimes the instruction can move the player into the ground, ie the player sinks into the ground, then continues to perform a bobbing action using the acceleration/gravity method, to fix this we perform a second trace after c_move, to see if the player has fallen into the ground, if it has we move the player up above the ground and set the z velocity to 0.

## Jumping

Jumping is a very broad area, there are many ways to do it and many techniques to create. To move the player up and down is rather simple, synchronizing and animation with a jump, getting the player to land from a fall, getting the player to animate a fall/land animation from simply stepping off a ledge takes a lot more work than jumping. You may want double jumping, wall jumping, all sorts of jumps. This is where your code may take on a more in depth perhaps messier approach. I admit this may is not the best framework to use for jumping, but it works. Movement scripts start out relatively simple and easy it is when you attempt to create the features of a movement script, wall jumping, ledge grabbing, ladder climbing, box carrying etc that the script becomes a bit of a hassle to create. The only features we will look into is combat and jumping, going further from there is up to you. If you can grasp the concepts and programming techniques explained in this tutorial, as well as using your own creativity, you could create anything that comes to mind. To put it in it's most simple form, creating features to a movement script is achieved by changing movement states of the player, have a variable, depending on the value of this variable have the player perform different movement, ie:

IF (movement_mode == 0) { handle_movement(); }
IF (movement_mode == 1) { handle_ladder_movement(); }

Handle_ladder_movement would be up and down movement, combined with a climbing animation. Then in your handle_movement(); script perform a trigger, trace or something to determine if the player has reached a ladder, if so switch the player's movement to ladder movement by setting movement_mode = 1; this may sound easy but implementing the movement to work smoothly, with out bugs and in sync with animations is the challenge. Time to implement jumping!

We'll keep this as simple as possible, we wont have any pause upon the player landing on the ground as the movement/combat style we're aiming for is fast paced and we want to give the player the feeling of cool interactive combat. Plus we don't really have a landing animation with our current model.

Make the following additions/changes to “handle_gravity”:

```
FUNCTION handle_gravity() {
      trace_mode = ignore_me+ignore_passable+use_box;
      result = trace(vector(my.x,my.y,my.z - my.z_offset),vector(my.x,my.y,my.z - 4000));

      IF (result < 3) {
            IF (my.jumping_mode == 0) {
                  my.force_z = -1 * result;
                  IF (key_space == 1 && my.animblend >= stand && my.animblend != jump &&
my.animblend != fall) {
                        my.jumping_mode = 1;
                        my.force_z = 25;
                        my.blendframe = jump;
                        my.animate2 = 0;
                        my.animblend = blend;
                  }
            }
            IF (my.jumping_mode == 2 || my.jumping_mode == 3) { my.jumping_mode = 0; }
      } ELSE {
            IF (my.jumping_mode == 2) {
                  IF (result > 120) {
                        my.animate = 60;
                        my.jumping_mode = 3;
                  } ELSE {
                        my.jumping_mode = 0;
                  }
            }
            IF (my.jumping_mode == 3 && result <= 120) { my.jumping_mode = 0; }
            IF (my.jumping_mode == 0) {
                  IF (result > 120 && my.animblend >= stand && my.animblend != jump &&
my.animblend != fall) {
                        my.jumping_mode = 3;
                        my.blendframe = fall;
                        my.animate2 = 0;
                        my.animblend = blend;
                  }
            }
            my.force_z -= 6 * time;
            my.force_z = max(-30,my.force_z);
      }

      my.velocity_z += (time * my.force_z) - (min(time*0.7,1) * my.velocity_z);
      my.move_z = my.velocity_z * time;
}
```

Make the following changes to “handle_animation”:

```
IF (my.currentframe == stand) { ent_animate(my,"stand",my.animate,anm_cycle); }
IF (my.currentframe == run) { ent_animate(my,"run",my.animate,anm_cycle); }
IF (my.currentframe == walk) { ent_animate(my,"walk",my.animate,anm_cycle); }
IF (my.currentframe == jump) { ent_animate(my,"jump",my.animate,0); }
IF (my.blendframe == stand) { ent_blend("stand",0,my.animate2); }
IF (my.blendframe == run) { ent_blend("run",0,my.animate2); }
IF (my.blendframe == walk) { ent_blend("walk",0,my.animate2); }
IF (my.blendframe == jump) { ent_blend("jump",0,my.animate2); }
IF (my.blendframe == fall) { ent_blend("jump",60,my.animate2); }
my.animate2 += 45 * time;
```

Further down in the function:

```
IF (my.animblend == walk) {
      ent_animate(my,"walk",my.animate,anm_cycle);
      my.animate += 8 * animation_speed * time;
      my.animate %= 100;
      my.currentframe = walk;
}
IF (my.animblend == jump || my.animblend == fall) {
      IF (my.jumping_mode == 3) { my.animate = 60; }
      ent_animate(my,"jump",my.animate,0);
      my.animate += 10 * animation_speed * time;
      my.currentframe = jump;
      IF (my.animate >= 60 && my.jumping_mode == 1) { my.jumping_mode = 2; }
      IF (my.animate >= 100) {
            ent_animate(my,"jump",100,0);
            my.animate = 100;
            my.blendframe = stand;
            IF (my.move_x != 0 || my.move_y != 0) {
                  IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe =
run; }
            }
      }
}
```

Make the following changes to “handle_movement”:

```
} ELSE {
      IF (my.animblend > stand && my.animblend != jump && my.animblend != fall) { //if we
aren't moving and our current animation is walk or run, blend and cycle the stand
animation
            my.blendframe = stand;
      }
}
```

The my.jumping_mode variable is the variable which handles jumping,

Run your code, press space bar to jump, you got a jumping animation! Congrats. Walk up a slope and fall off at a high height, the player plays a falling animation :D

```
IF (result < 3) {
      IF (my.jumping_mode == 0) {
            my.force_z = -1 * result;
            IF (key_space == 1 && my.animblend >= stand && my.animblend != jump &&
my.animblend != fall) {
                  my.jumping_mode = 1;
                  my.force_z = 25;
                  my.blendframe = jump;
                  my.animate2 = 0;
                  my.animblend = blend;
            }
      }
```

If we are on the ground (result < 3), and we aren't already jumping or falling (my.jumping_mode == 0), if we have pressed space bar and we are either playing the stand, walk or run animation (not jump or fall), begin the jump “my.jumping_mode = 1”, apply upwards force “my.force_z = 25”, set animation to jumping “my.blendframe = jump;”, normally with controlling animations this would be the only instruction needed, but because “handle_movement” after this there is a possibility that my.blendframe may change to stand/walk or run whilst jump is still in the blending process, so we have gone ahead and used “my.animate2 = 0; my.animblend = blend;” to stop this potential bug.

This code handles 2 things, falling and jumping, fall occurs when you step off a ledge and do not perform a jump, jumping occurs when you press the space bar. Both animations/modes use the jump animation, except when we fall off a ledge rather than playing the jump animation we blend from our current animation to half way through the jump animation (60%), this means rather than blending to the beginning of the jump animation we blend to 60% of it, this way the initial jumping motion of the animation is not performed. Let's look at our animation function now:

```
IF (my.blendframe == jump) { ent_blend("jump",0,my.animate2); }
IF (my.blendframe == fall) { ent_blend("jump",60,my.animate2); }
```

The fall animation works and uses the jump animation, but simply starts the animation differently. As you can see there is a value of 60 in the blend function to blend to 60% of the jump animation.

```
IF (my.animblend == jump || my.animblend == fall) {
      IF (my.jumping_mode == 3) { my.animate = 60; }
      ent_animate(my,"jump",my.animate,0);
      my.animate += 10 * animation_speed * time;
      my.currentframe = jump;
      IF (my.animate >= 60 && my.jumping_mode == 1) { my.jumping_mode = 2; }
      IF (my.animate >= 100) {
            ent_animate(my,"jump",100,0);
            my.animate = 100;
            my.blendframe = stand;
            IF (my.move_x != 0 || my.move_y != 0) {
                  IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe = 
run; }
            }
      }
}
```

If blendframe == jump or fall we will play the jump animation. If (jumping_mode == 3) this can mean 2 things, we have jumped and at the height of our jump have a long way to fall so hold the animation in position until we are close enough to the ground to finish it. It also means we have stepped off a ledge that is high enough in the air to perform the fall animation. While jumping_mode == 3 my.animate is set to 60, a freeze in the jump animation. Also note that in all ent_animate functions, including the one in the blend section, we have 0 in replace of anm_cycle, this is because our jump animation is non cyclic, we want it to play from 0-100 and not repeat itself.

If (my.animate >= 60) and if we have performed a jump, set the jumping mode to 2. This is to perform a check whether or not to continue playing the jump animation or freeze it as the player has some distance to fall. After my.jumping_mode is set to 2 have a look at what happens over in your “handle_gravity” function:

After the jump animation has completed, we either blend to the stand/walk or run animation depending on whether the player is moving or not, and holding shift.

```
IF (my.jumping_mode == 2) {
      IF (result > 120) {
            my.animate = 60;
            my.jumping_mode = 3;
      } ELSE {
            my.jumping_mode = 0;
      }
}
IF (my.jumping_mode == 3 && result <= 120) { my.jumping_mode = 0; }
```

This code is in the else section of (result < 3), this means we are above the ground, If (result > 120) if we have a long way to fall, a distance greater than 120 quants set our animation to freeze at 60 until we are close enough to the ground, otherwise set my.jumping_mode to 0 to finish the jumping animation. If we are falling (my.jumping_mode == 3) and we are close enough to the ground (result <= 120), finish off the jump animation “my.jumping_mode = 0;”

```
IF (my.jumping_mode == 2 || my.jumping_mode == 3) { my.jumping_mode = 0; }
```

This code sets our my.jumping_mode to 0, if we are on the ground and my.jumping_mode is falling (3) or deciding on whether to fall or not (2).

```
IF (my.jumping_mode == 0) {
      IF (result > 120 && my.animblend >= stand && my.animblend != jump && my.animblend !=
fall) {
            my.jumping_mode = 3;
            my.blendframe = fall;
            my.animate2 = 0;
            my.animblend = blend;
      }
}
```

This code begins the falling of the player, notice again it is in the else section of (result < 3), it needs to be to make sure your character/result is above ground. If you are not jumping, and the distance to the ground is greater than 120, and you are not already animating jump/fall and you are animating stand/walk or run, begin falling by setting my.blendframe = fall. You also set my.jumping_mode = 3; to change into falling mode.

I hope this isn't too confusing, this is where it starts to get complicated as you are linking variables in various functions together to get animations working properly in sync with movement. There may be better ways to set this up, and this may not be the best framework to use. It's a good start though and it certainly works :)

If you hold down space bar your player will continually jump, what if we want it that you must release space bar before being able to jump again?

Add the following variable at the top of your script:

```
var space_press = 0;
```

In your “handle_gravity” function find the following code:

```
IF (key_space == 1 && my.animblend >= stand && my.animblend != jump && my.animblend !=
fall) {
```

And replace with:

```
IF (key_space == 0 && space_press == 1) { space_press = 0; }
IF (key_space == 1 && space_press == 0 && my.animblend >= stand && my.animblend != jump &&
my.animblend != fall) {
      space_press = 1;
```

This code ensures that you can only press space bar when (space_press == 0), after performing a jump space_press is set to 1, and is only set back to 0 if you are not pressing space.

**Camera Collision**

I will be showing you two ways to create camera collision, one is slightly smoother than the other, the more rigid one would be more useful in a shooter environment, easier to control the cross hair.

*Stable/rigid camera collision*

Make the following changes to “handle_camera”:

```
temp = fcos(camera.tilt,-camera_distance);
vec_set(camera.x,vector(my.x + fcos(camera.pan,temp),my.y + fsin(camera.pan,temp),my.z +
20 + fsin(camera.tilt,-camera_distance)));

vec_diff(temp.x,camera.x,my.x);
vec_normalize(temp.x,16);
vec_add(temp.x,camera.x);

trace_mode = ignore_me+ignore_passable+ignore_models;
result = trace(my.x,temp.x);
IF (result > 0) {
      vec_diff(temp.x,my.x,target.x);
      vec_normalize(temp.x,16);
      vec_set(camera.x,target.x);
      vec_add(camera.x,temp.x);
}
```

Run your level and you now have working camera collision!  A trace instruction is being used, from the player to the camera, if the trace has hit a wall (result > 0) then the camera is placed at the position the trace hit.  To explain all the vector instructions that are going on, our camera collision in it's most simple form could be coded like this:

```
trace_mode = ignore_me+ignore_passable;
result = trace(my.x,camera.x);
IF (result > 0) { vec_set(camera.x,target.x); }
```

What we are doing above is tracing 16 quants behind the camera,  if the trace hits something within these 16 quants the camera is placed at the target position then moved 16 quants towards the player.  It works just the same as the smaller version above, but it will keep the camera further away from the wall and avoid clipping. You may want to add “camera.clip_far = 0;” at the beginning of your main script to avoid being able to see through objects that the camera is very close to.  If you would like to understand what exactly is going on with the vectors in this code please see the example code that came with this tutorial, the function is commented out as the example uses smooth camera but there are comment tags explaining what each vector function is performing.

*Smooth camera collision*

Replace the “handle_camera” function with the following (including new variables):

```
var camera_move_to[3];
var camera_pan;
var camera_tilt;

FUNCTION handle_camera() {
      camera_pan -= mouse_force.x * 12 * time;
      camera_tilt += mouse_force.y * 8 * time;
      camera_tilt = clamp(camera_tilt,-30,10);

      camera.pan = camera_pan;
      temp = fcos(camera_tilt,-camera_distance);
      vec_set(camera_move_to.x,vector(my.x + fcos(camera.pan,temp),my.y +
fsin(camera.pan,temp),my.z + 20 + fsin(camera_tilt,-camera_distance)));

      //this is similar code to the template camera, the camera needs to be at the
camera_move_to coordinates, this traces at a position, which smoothly moves towards the
camera_move_to, and places the camera at the target.x vector, this works well because
instead of placing the camera to camera_move_to, it traces at a position which smoothly
moves to camera_move_to
      // move towards target position
      temp = min(1,0.5 * time);    // value of 1 places us at target, this value is what
allows the smooth movement
      camera.x += temp*(camera_move_to.x - camera.x);
      camera.y += temp*(camera_move_to.y - camera.y);
      camera.z += temp*(camera_move_to.z - camera.z);

      // keep camera from penetrating walls
      vec_diff(temp.x,camera.x,my.x);
      vec_normalize(temp.x,16);
      vec_add(temp.x,camera.x);

      trace_mode = ignore_me + ignore_passable + ignore_models;
      IF (trace(my.x,temp.x) > 0) {
            vec_diff(temp.x,my.x,target.x);
            vec_normalize(temp.x,16);
            vec_set(camera.x,target.x);
            vec_add(camera.x,temp.x);
      } ELSE {
            vec_set(camera.x,camera.x);
      }

      vec_diff(temp.x,my.x,camera.x);
      vec_to_angle(camera.pan,temp.x);
}
```

This code works similar to the rigid camera code except for a few extra lines. Firstly we are adding camera_pan and camera_tilt variables in replace of camera.tilt and camera.pan in regards to using the mouse to rotate and move the camera around. In order to use smooth movement of the camera, rather than setting linking the camera's position directly to the camera's pan and tilt, we are linking it's movement to the camera's pan and tilt, but linking it's pan and tilt to directly face the player. I hope that doesn't sound too confusing. Try changing camera_pan to camera.tilt, and camera_tilt to camera.tilt, then comment out the last 2 lines of code setting the camera's angle, the camera kinda behaves how it does in Axys where the camera isn't perfectly centered on the player, but as you rotate the camera left and right the player moves to the left and right of the screen, it may take tweaking to get this to work right.

```
temp = min(1,0.5 * time);    // value of 1 places us at target, this value is what allows
the smooth movement
camera.x += temp*(camera_move_to.x - camera.x);
camera.y += temp*(camera_move_to.y - camera.y);
camera.z += temp*(camera_move_to.z - camera.z);
```

The camera smoothly moves to the camera_move_to position, the further the camera is from the camera_move_to position the more distance it moves for each frame, this gives the feeling of a smooth move to, as the closer the camera is to the position it needs to move to the slower it moves. The "temp = min(1,0.5 * time);" instruction is how smooth the camera moves, it also stops weird behavior of the camera if the frame rate drops too low but not allowing temp to exceed 1 (ie fps less than 32). If you would like to change how smooth the camera moves, change the 0.5 value, if you set 0.5 to 1 the camera will work just like rigid, at 0.1 the camera will move very slowly.

That's it for camera collision. Finally it's time for combat.

### Weapon Before Combat

Let's give your guy a weapon. Add the following code above "player_action":

```
var temp2[6];

ACTION attach_weapon {
      my.passable = on;
      proc_late();
      WHILE (you != null) {
            vec_for_vertex(temp.x,you,1175); //hand palm base
            vec_for_vertex(temp2.x,you,1240); //hand palm tip
            vec_set(my.x,temp.x);
            vec_diff(temp.x,temp2.x,temp.x);
            vec_to_angle(temp.pan,temp.x);
            vec_set(my.pan,temp.pan);
            wait(1);
      }
}
```

We are adding a new vector variable temp2, just the same as temp but it comes in handy using two or even three vectors for temporary calculations. Make the following changes to "player_action":

```
ACTION player_action {
      my.z_offset = 6;
      my.shadow = on;
      ent_create("sword.mdl",my.x,attach_weapon);
      wait(1);
      WHILE (1) { //the main loop
```

Run your script and watch as your player runs around with a wicked cool sword :) I'm assuming you understand the ent_create instruction, read the 3DGS manual if you don't.

```
my.passable = on;
proc_late();
```

We are making the weapon passable so that the player and trace instructions do not collide with it. Proc_late places the action at the end of the function list (ie it makes sure that this function is processed after the player function has been), this ensures that the player's movement is calculated first then the sword is placed at the hand

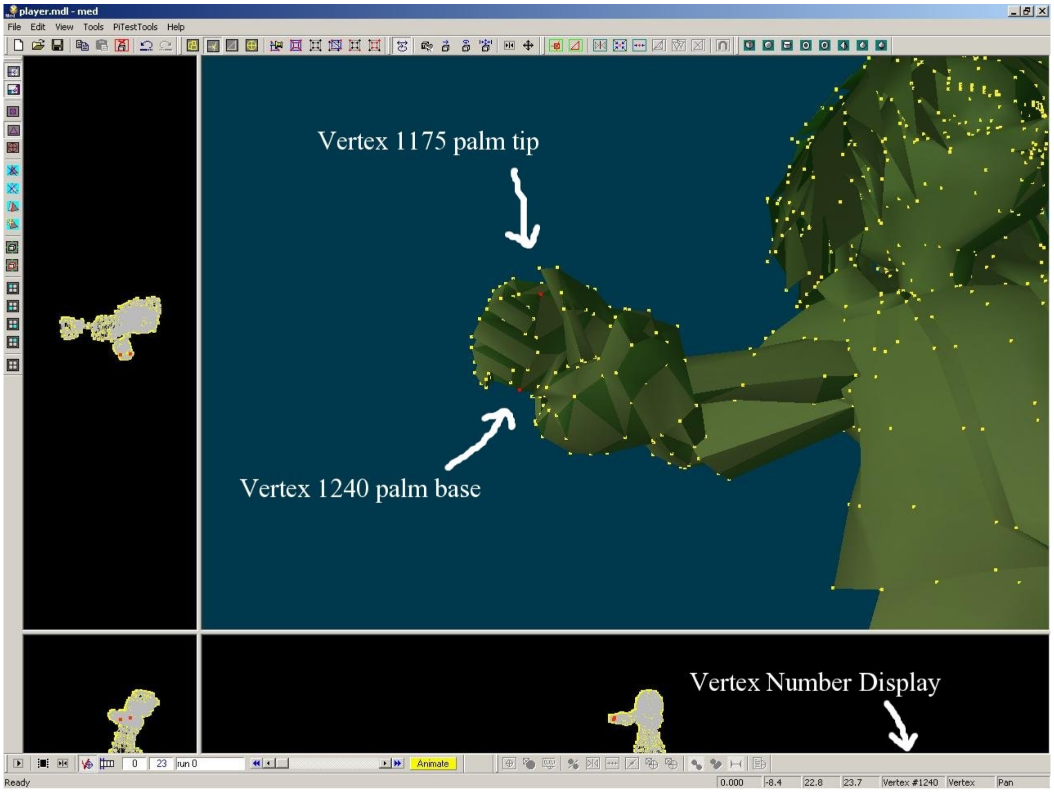
and not the other way around, if the sword were to be placed at the hand position first then the player's movement changed, the sword would be at the position the player's hand was at the previous frame and not the current one.

```
WHILE (you != null) {
```

I am also assuming you understand the you pointer and how that is used.  This line ensures that the player in the form of a you pointer still exists and can be accessed by this function so that we don't get a non existent or empty pointer error.

```
vec_for_vertex(temp.x,you,1175); //hand palm base
vec_for_vertex(temp2.x,you,1240); //hand palm tip
```

Here we are storing the world coordinates of the player's right hand palm base and tip, this is because we want to place the sword at the coordinate of the palm base, and angle the sword based on the vector from the palm base to the palm tip, so that the sword has the same alignment as the hand.



This is how I found which vertex points to use, I opened up med and selected the vertex at the base of the hand, looked at the bottom of the screen to find it was vertex 1240, I did the same for the top of the palm.  Now in code I can access the correct verticies and know that I'm using the correct ones.

```
vec_set(my.x,temp.x);
vec_diff(temp.x,temp2.x,temp.x);
vec_to_angle(temp.pan,temp.x);
vec_set(my.pan,temp.pan);
```

Set the coordinates of the sword to the same coordinates at the base of the palm.  Then find the vector from the base of the palm to the top, and get the angle of this vector, align the swords angle to the angle of this vector.

Please note how the sword is set up in MED, this technique for attaching weapons is very good and efficient compared to animating the sword separately. This means you can attach any weapon you have a model of to the player's hand. You just need to make sure the model is placed and rotated properly in regards to the origin of MED for the sword to work well. Noticed that the hilt of the sword is at the origin Also notice the way the blade is tilted, if it were rotated around the sword would also be rotated that direction in the world unless coded or changed otherwise.

You add the wait(1); instruction after creating the entity, because when it is first created it is created with it's passable flag set to off, this means the first trace for gravity will hit it and the player may fly up in some weird place. So we wait the one frame for the sword's action to begin and set itself to passable before beginning the player's action.

# COMBAT

### A Single Attack

I'm sure you've been waiting a long time to get to this. Combat is surely the most satisfying part of game play, it is very fun to run around and whack some baddies with an awesome sword :)

Our player has 6 attacks, which we can chain into a number of combos however we please, along with airborne attacks. Let's start by coding a single attack. We need to add some defines to our animations, go up to the top of your script where you have your animation defines and add the following:

```
DEFINE attack_a,5;
DEFINE attack_b,6;
DEFINE attack_c,7;
DEFINE attack_d,8;
DEFINE attack_e,9;
DEFINE attack_f,10;
```

Add the following variable:

```
var space_press = 0;
var mouse_left_press = 0;
```

Make the following changes to “handle_movement”:

```
IF (my.movement_mode == 0) {
    IF (my.move_x != 0 || my.move_y != 0) { //if we are moving
        IF (my.animblend == stand) { //if our current animation is stand
            IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe =
run; }
        }
        IF (my.animblend == run && key_shift == 1) { my.blendframe = walk; }
        IF (my.animblend == walk && key_shift == 0) { my.blendframe = run; }
    } ELSE {
        IF (my.animblend > stand && my.animblend != jump && my.animblend != fall)
{ //if we aren't moving and our current animation is walk or run, blend and cycle the
stand animation
            my.blendframe = stand;
        }
    }
    IF (mouse_left == 0 && mouse_left_press == 1) { mouse_left_press = 0; }
    IF (mouse_left == 1 && mouse_left_press == 0 && my.animblend >= stand) {
        mouse_left_press = 1;
        my.blendframe = attack_a;
        my.movement_mode = 1;
    }
}
IF (my.movement_mode == 1) {
    my.jumping_mode = 0;
}
```

Make the following changes to “handle_gravity”:

```
IF (key_space == 1 && space_press == 0 && my.movement_mode == 0 && my.animblend >= stand
&& my.animblend != jump && my.animblend != fall) {
```

Further down in “handle_gravity”:

```
IF (my.jumping_mode == 3 && result <= 120) { my.jumping_mode = 0; }
IF (my.jumping_mode == 0 && my.movement_mode == 0) {
    IF (result > 120 && my.animblend >= stand && my.animblend != jump && my.animblend !=
fall) {
```

Make the following changes to “handle_animation”:

```
IF (my.currentframe == jump) { ent_animate(my,"jump",my.animate,0); }
IF (my.currentframe == attack_a) { ent_animate(my,"attack_a",my.animate,0); }
IF (my.blendframe == stand) { ent_blend("stand",0,my.animate2); }
IF (my.blendframe == run) { ent_blend("run",0,my.animate2); }
IF (my.blendframe == walk) { ent_blend("walk",0,my.animate2); }
IF (my.blendframe == jump) { ent_blend("jump",0,my.animate2); }
IF (my.blendframe == fall) { ent_blend("jump",60,my.animate2); }
IF (my.blendframe == attack_a) { ent_blend("attack_a",0,my.animate2); }
my.animate2 += 45 * time;
```

Further down in the function add the following:

```
IF (my.animblend == attack_a) {
      ent_animate(my,"attack_a",my.animate,0);
      my.animate += 20 * animation_speed * time;
      my.currentframe = attack_a;
      IF (my.animate >= 100) {
            my.movement_mode = 0;
            ent_animate(my,"attack_a",100,0);
            my.animate = 100;
            my.blendframe = stand;
            IF (my.move_x != 0 || my.move_y != 0) {
                  IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe =
run; }
            }
      }
}
```

Run your code, left click to attack. Our player attacks! Nothing impressive, but it's still cool.

```
IF (my.movement_mode == 0) {
...
IF (mouse_left == 0 && mouse_left_press == 1) { mouse_left_press = 0; }
IF (mouse_left == 1 && mouse_left_press == 0 && my.animblend >= stand) {
      mouse_left_press = 1;
      my.blendframe = attack_a;
      my.movement_mode = 1;
}
```

If the player is not attacking (my.movement_mode == 0) we want the walk/run/stand animations to be in progress, but if the player is attacking we don't want to begin any of these animations. If the player presses mouse_left (mouse_left == 1) and the current animation frame is anything except blending (my.animblend >= stand) then begin the attack animation “my.blendframe = attack_a;”, and set my.movement_mode to 1 to ensure that no other animations begin while we are attacking. mouse_left_press ensures you must release left click before being able to attack again.

```
IF (my.movement_mode == 1) {
      my.jumping_mode = 0;
}
```

As my.jumping_mode only exists for the purpose of playing jump animations and is not part of the jumping itself, we want to turn off jumping animations “my.jumping_mode = 0;” if we are attacking (my.movement_mode == 1).

```
IF (key_space == 1 && space_press == 0 && my.movement_mode == 0 && my.animblend >= stand
&& my.animblend != jump && my.animblend != fall) {
```

This ensures that jumping animation cannot begin if we are attacking.

```
IF (my.jumping_mode == 3 && result <= 120) { my.jumping_mode = 0; }
IF (my.jumping_mode == 0 && my.movement_mode == 0) {
      IF (result > 120 && my.animblend >= stand && my.animblend != jump && my.animblend !=
fall) {
```

This ensures that falling animation cannot begin if we are attacking.

```
IF (my.currentframe == attack_a) { ent_animate(my,"attack_a",my.animate,0); }
IF (my.blendframe == attack_a) { ent_blend("attack_a",0,my.animate2); }
```

As the attack animation is non cyclic a 0 is at the end of the ent_animate function rather than anm_cycle.

```
IF (my.animblend == attack_a) {
      ent_animate(my,"attack_a",my.animate,0);
      my.animate += 20 * animation_speed * time;
      my.currentframe = attack_a;
      IF (my.animate >= 100) {
            my.movement_mode = 0;
            ent_animate(my,"attack_a",100,0);
            my.animate = 100;
            my.blendframe = stand;
            IF (my.move_x != 0 || my.move_y != 0) {
                  IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe =
run; }
            }
      }
}
```

This is playing through the attack animation just like the jump animation.  Once the animation has completed the player's animation is set back to either stand/run or walk depending on whether the player is moving and holding shift. Attacking mode is canceled “my.movement_mode = 0;”

Time to implement combos.

**Combos**

Add this variable at the top of your script:

```
var combo_continue = 0;
```

Make the following changes to “handle_movement”:

```
      IF (mouse_left == 1 && mouse_left_press == 0 && my.animblend >= stand) {
            mouse_left_press = 1;
            my.blendframe = attack_a;
            my.movement_mode = 1;
            combo_continue = 0;
      }
}
IF (my.movement_mode == 1) {
      my.jumping_mode = 0;
      IF (mouse_left == 0 && mouse_left_press == 1) { mouse_left_press = 0; }
      IF (mouse_left == 1 && mouse_left_press == 0 && my.animate >= 30 && combo_continue
== 0) { mouse_left_press = 1; combo_continue = 1; }
}
```

Make the following changes to “handle_animation”:

```
IF (my.currentframe == attack_a) { ent_animate(my,"attack_a",my.animate,0); }
IF (my.currentframe == attack_b) { ent_animate(my,"attack_b",my.animate,0); }
IF (my.currentframe == attack_c) { ent_animate(my,"attack_c",my.animate,0); }
IF (my.currentframe == attack_d) { ent_animate(my,"attack_d",my.animate,0); }
IF (my.currentframe == attack_e) { ent_animate(my,"attack_e",my.animate,0); }
IF (my.currentframe == attack_f) { ent_animate(my,"attack_f",my.animate,0); }


IF (my.blendframe == attack_a) { ent_blend("attack_a",0,my.animate2); }
IF (my.blendframe == attack_b) { ent_blend("attack_b",0,my.animate2); }
IF (my.blendframe == attack_c) { ent_blend("attack_c",0,my.animate2); }
IF (my.blendframe == attack_d) { ent_blend("attack_d",0,my.animate2); }
IF (my.blendframe == attack_e) { ent_blend("attack_e",0,my.animate2); }
IF (my.blendframe == attack_f) { ent_blend("attack_f",0,my.animate2); }


IF (my.animblend == attack_a) {
      ent_animate(my,"attack_a",my.animate,0);
      my.animate += 20 * animation_speed * time;
      my.currentframe = attack_a;
      IF (my.animate >= 100) {
            my.movement_mode = 0; //important, remove this line
            ent_animate(my,"attack_a",100,0);
            my.animate = 100;
            my.blendframe = stand;
            IF (my.move_x != 0 || my.move_y != 0) {
                  IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe = 
run; }
            }
            IF (combo_continue == 1) {
                  combo_continue = 0;
                  my.blendframe = attack_b;
            } ELSE {
                  my.movement_mode = 0;
            }
      }
}
```

```
IF (my.animblend == attack_b) {
      ent_animate(my,"attack_b",my.animate,0);
      my.animate += 15 * animation_speed * time;
      my.currentframe = attack_b;
      IF (my.animate >= 100) {
            ent_animate(my,"attack_b",100,0);
            my.animate = 100;
            my.blendframe = stand;
            IF (my.move_x != 0 || my.move_y != 0) {
                  IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe =
run; }
            }
            IF (combo_continue == 1) {
                  combo_continue = 0;
                  my.blendframe = attack_c;
            } ELSE {
                  my.movement_mode = 0;
            }
      }
}

IF (my.animblend == attack_c) {
      ent_animate(my,"attack_c",my.animate,0);
      my.animate += 10 * animation_speed * time;
      my.currentframe = attack_c;
      IF (my.animate >= 100) {
            ent_animate(my,"attack_c",100,0);
            my.animate = 100;
            my.blendframe = stand;
            IF (my.move_x != 0 || my.move_y != 0) {
                  IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe =
run; }
            }
            IF (combo_continue == 1) {
                  combo_continue = 0;
                  my.blendframe = attack_d;
            } ELSE {
                  my.movement_mode = 0;
            }
      }
}
```

```
IF (my.animblend == attack_d) {
      ent_animate(my,"attack_d",my.animate,0);
      my.animate += 15 * animation_speed * time;
      my.currentframe = attack_d;
      IF (my.animate >= 100) {
            ent_animate(my,"attack_d",100,0);
            my.animate = 100;
            my.blendframe = stand;
            IF (my.move_x != 0 || my.move_y != 0) {
                  IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe =
run; }
            }
            IF (combo_continue == 1) {
                  combo_continue = 0;
                  my.blendframe = attack_e;
            } ELSE {
                  my.movement_mode = 0;
            }
      }
}

IF (my.animblend == attack_e) {
      ent_animate(my,"attack_e",my.animate,0);
      my.animate += 15 * animation_speed * time;
      my.currentframe = attack_e;
      IF (my.animate >= 100) {
            ent_animate(my,"attack_e",100,0);
            my.animate = 100;
            my.blendframe = stand;
            IF (my.move_x != 0 || my.move_y != 0) {
                  IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe =
run; }
            }
            IF (combo_continue == 1) {
                  combo_continue = 0;
                  my.blendframe = attack_f;
            } ELSE {
                  my.movement_mode = 0;
            }
      }
}
```

```
IF (my.animblend == attack_f) {
      ent_animate(my,"attack_f",my.animate,0);
      my.animate += 6 * animation_speed * time;
      my.currentframe = attack_f;
      IF (my.animate >= 100) {
            my.movement_mode = 0;
            ent_animate(my,"attack_f",100,0);
            my.animate = 100;
            my.blendframe = stand;
            IF (my.move_x != 0 || my.move_y != 0) {
                  IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe =
run; }
            }
      }
}
```

Remember to remove the “my.movement_mode = 0;” that you originally had after “IF (my.animate >= 100) {“ in the attack_a if statement, it is highlighted in bold red text in the above code. That is now being set in the else part of combo_continue. Otherwise the combos wont work, the combo will always end after the first attack.

Run your code, as you continually click the left mouse button your character will perform combos.

```
IF (my.movement_mode == 1) {
      my.jumping_mode = 0;
      IF (mouse_left == 0 && mouse_left_press == 1) { mouse_left_press = 0; }
      IF (mouse_left == 1 && mouse_left_press == 0 && my.animate >= 30 && combo_continue
== 0) { mouse_left_press = 1; combo_continue = 1; }
}
```

While our character is attacking (my.movement_mode == 1) if we click the left mouse button (mouse_left == 1) and if we are at least 30% into the attack animation (my.animate >= 30) set the combo variable “combo_continue = 1;” when combo_continue is set to 1 this means to perform the next attack in the combo rather than stopping the combo all together. The animation code for all of your new attacks is the same as the code you used for the single attack. This time however you will notice a new instruction which determines whether to continue onto a new combo animation or end the attack:

```
IF (combo_continue == 1) { combo_continue = 0; my.blendframe = attack_b; my.movement_mode
= 1; }
IF (combo_continue == 1) {
      combo_continue = 0;
      my.blendframe = attack_b;
} ELSE {
      my.movement_mode = 0;
}
```

This code appears in every attack animation except for the last one where there is no continued attack. If you have clicked the left mouse button sometime after 30% into the attack animation (combo_continue == 1), move onto the next attack animation:

If combo_continue is equal to 0 the current attack will end. Otherwise combo_continue is set to 0 and the combo continues to the next animation.

So now we have some rough working combos. We will soon work on refining it more, as well as adding some dummy enemies. At the moment the player may have a little too much control over the character whilst attacking, you can move in any direction and stop. Let's start by not allowing the main character to stop moving whilst attacking. Also let's get some airborne movement involved in some of those cool uppercut type attack animations.

### Combo Refinement

Firstly let's make the character constantly move forward whilst attacking, and move forward at various speeds to make it feel good (ie faster movement during the middle of the attack and slower towards the end).

Make the following changes to “handle_movement”:

```
IF (temp.x != -1000) {
	IF (key_shift == 1) { temp.y = 10 * time; } ELSE { temp.y = 15 * time; }
}
IF (my.movement_mode == 0) {
	my.move_x = fcos(temp.x,temp.y);
	my.move_y = fsin(temp.x,temp.y);
}
IF (my.movement_mode == 1) {
	temp.y = fsin((my.animate * 1.2) + 45,15 * time);
	my.move_x = fcos(my.pan,temp.y);
	my.move_y = fsin(my.pan,temp.y);
	temp.y = 0;
	IF (temp.x != -1000) { temp.y = 1; }
}
c_move(my,nullvector,my.move_x,use_aabb | ignore_passable | glide);
```

Run your script and watch as the amount of movement the player performs for each attack feels more realistic, and a little cooler.

```
IF (my.movement_mode == 0) {
	my.move_x = fcos(temp.x,temp.y);
	my.move_y = fsin(temp.x,temp.y);
}
```

This is the normal movement used for running around, we only want to use this movement if we aren't attacking.

```
IF (my.movement_mode == 1) {
	temp.y = fsin((my.animate * 1.2) + 45,15 * time);
	my.move_x = fcos(my.pan,temp.y);
	my.move_y = fsin(my.pan,temp.y);
```

If we are attacking (my.movement_mode == 1) then we want to move the player only in the direction it is facing, that is why you see my.pan in the fcos and fsin functions rather than temp.x, the player is still able to turn and rotate the character but the character will never stop moving and always move forward.

The amount of distance the player moves is calculated in a sin function. If you don't understand the mathematics behind this function don't worry about it. When my.animate == 0 the player will move at a speed of 7.5 * time, my.animate == 37.5 (almost half way through the animation) the player will be moving at a top speed of 15 * time, when my.animate == 100 the player will move at a speed of 2.5 * time, so it's a smooth change in speed throughout the animation. I simply tweaked with the values until the character behaved in a way that was realistic enough for me to be happy with. I know how daunting such equations can be to the beginner, or one who doesn't understand it. I do encourage you to find some simple tutorials on trigonometry, once you get a good understanding of it see how you can use the functions to create smooth movement. I have found sin and cos functions to be the most important feature of creating smooth movement, with the camera and the player. I have recently be working on a turn based RPG, and have used sin and cos functions to move the player between itself and the enemy as it performs a jump animation, then it performs the attack and does the opposite backwards. I have also used them in creating smooth camera introductions to the battle.

```
temp.y = 0;
IF (temp.x != -1000) { temp.y = 1; }
```

Here you are setting temp.y to 0, as soon after this instruction you will find:

```
IF (temp.y > 0) { rotate_entity(temp.x,30); }
```

You don't want to rotate the character unless the player is pressing WSAD to move the character in another direction, otherwise the character would constantly rotate towards whatever value temp.x is. If the player is pressing a button (temp.x != -1000), then rotate the player towards the direction the keys are being pressed "temp.y = 1;"

Time to get our player to fly upwards with some of those attacks. We will add a new value to my.jumping_mode, to be used when the player will move upwards but doesn't require animation change. At the moment if we simply added a "my.force_z = 30;" statement as our attacks begin, the force would immediately be reset by our gravity function because we are too close to the ground:

```
IF (result < 3) {
      IF (my.jumping_mode == 0) {
            my.force_z = -1 * result;
```

We need to set my.jumping_mode to a value like 10, then set it back to 0 once the player is high enough off the ground, or as we have done with jumping, once the animation has progressed pass a certain point.

Make the following changes to "handle_movement":

```
IF (my.movement_mode == 1) {
      IF (my.jumping_mode != 10) {
            my.jumping_mode = 0;
      } ELSE {
            IF (my.animate >= 60 && my.animblend >= stand) { my.jumping_mode = 0; }
      }
      IF (mouse_left == 0 && mouse_left_press == 1) { mouse_left_press = 0; }
      IF (mouse_left == 1 && mouse_left_press == 0 && my.animate >= 30 && combo_continue
== 0) { mouse_left_press = 1; combo_continue = 1; }
}
```

Make the following changes to "handle_animation":

```
IF (my.animblend == attack_c) {
      ent_animate(my,"attack_c",my.animate,0);
      my.animate += 10 * animation_speed * time;
      my.currentframe = attack_c;
      IF (my.animate >= 100) {
            my.movement_mode = 0;
            ent_animate(my,"attack_c",100,0);
            my.animate = 100;
            my.blendframe = stand;
            IF (my.move_x != 0 || my.move_y != 0) {
                  IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe =
run; }
            }
            IF (combo_continue == 1) {
                  my.jumping_mode = 10;
                  my.force_z = 12;
                  combo_continue = 0;
                  my.blendframe = attack_d;
            } ELSE {
                  my.movement_mode = 0;
            }
      }
}

IF (my.animblend == attack_e) {
      ent_animate(my,"attack_e",my.animate,0);
      my.animate += 15 * animation_speed * time;
      my.currentframe = attack_e;
      IF (my.jumping_mode == 0 && my.animate >= 20 && my.animate < 60) {
            my.jumping_mode = 10;
            my.force_z = 20;
      }
      IF (my.animate >= 100) {
            my.movement_mode = 0;
            ent_animate(my,"attack_e",100,0);
            my.animate = 100;
            my.blendframe = stand;
            IF (my.move_x != 0 || my.move_y != 0) {
                  IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe =
run; }
            }
            IF (combo_continue == 1) {
                  combo_continue = 0;
                  my.blendframe = attack_f;
            } ELSE {
                  my.movement_mode = 0;
            }
      }
}
```

```
IF (my.animblend == attack_f) {
      ent_animate(my,"attack_f",my.animate,0);
      my.animate += 6 * animation_speed * time;
      my.currentframe = attack_f;
      IF (my.jumping_mode == 0 && my.animate >= 40 && my.animate < 60) {
            my.jumping_mode = 10;
            my.force_z = 12;
      }
      IF (my.animate >= 100) {
            my.movement_mode = 0;
            ent_animate(my,"attack_f",100,0);
            my.animate = 100;
            my.blendframe = stand;
            IF (my.move_x != 0 || my.move_y != 0) {
                  IF (key_shift == 1) { my.blendframe = walk; } ELSE { my.blendframe =
run; }
            }
      }
}
```

At the very end of your "handle_animation" function add the following line:

```
IF (my.animblend != blend && my.blendframe != nullframe) { my.animate2 = 0; my.animblend =
blend; }
```

Run your script, now your player goes in the air with some of those attacks.

```
IF (my.movement_mode == 1) {
      IF (my.jumping_mode != 10) {
            my.jumping_mode = 0;
      } ELSE {
            IF (my.animate >= 60 && my.animblend >= stand) { my.jumping_mode = 0; }
      }
      IF (mouse_left == 0 && mouse_left_press == 1) { mouse_left_press = 0; }
      IF (mouse_left == 1 && mouse_left_press == 0 && my.animate >= 30 && combo_continue
== 0) { mouse_left_press = 1; combo_continue = 1; }
}
```

If the character's jumping mode is anything but 10, set jumping_mode to 0 (normal gravity mode) else if the jumping_mode is 10, this means our character is jumping off the ground in a combo, if the animation is greater than 60% and if we aren't blending, set jumping_mode/gravity back to normal. This is to stop the part of the gravity function (result < 3) being processed when the player initially launches off the ground, otherwise the force would be reset "my.force_z = -1 * result;"

```
IF (combo_continue == 1) {
      my.jumping_mode = 10;
      my.force_z = 12;
      combo_continue = 0;
      my.blendframe = attack_d;
} ELSE {
      my.movement_mode = 0;
}
```

At the end of two of your combo animations you are setting jumping mode without animation to on (this means the player will use the else side of the gravity until the animation exceeds 60% "my.jumping_mode = 10;" and you are applying an initial z force "my.force_z = 12;" This works well for some of your attacks, but for the final one you may not want the player going airborne straight away as it doesn't line up with the animation well. So instead you have the player going airborne after 40% into the animation:

```
IF (my.jumping_mode == 0 && my.animate >= 40 && my.animate < 60) {
      my.jumping_mode = 10;
      my.force_z = 12;
}
```

You need to make sure that you haven't already gone airborne “my.jumping_mode == 0“ otherwise my.force_z would continually be set to 12 and you would fly up into the air much higher than needed. You also need to make sure this wont occur once the animation exceeds 60% “my.animate < 60” as a that point my.jumping_mode is set to 0 in “handle_movement” and you don't want the force and my.jumping_mode being set again after the animation exceeds 60. You are also using the same technique for previous attack, this way you can get the jumping to occur a little later in your animation, having more control over how your combo works.

```
IF (my.animblend != blend && my.blendframe != nullframe) { my.animate2 = 0; my.animblend =
blend; }
```

As you are changing the animation at the end of each combo, you need to make sure you set it to blend before the next frame finishes (this instruction is also found at the top of handle_animation), if you don't set it to blend before the next frame then the following instruction will be processed in “handle_movement”:

```
IF (my.animate >= 60 && my.animblend >= stand) { my.jumping_mode = 0; }
```

This would mean the jumping mode you set at the end of your attack_c combo (for the attack_d jump) would not occur as my.jumping_mode is set to 0 rather than 10, and my.force_z would be set at “my.force_z = -1 * result;” in the gravity function.

This looks pretty cool. It would look better if the gravity was a bit slower on some of your attacks. Let's introduce a gravity variable, that when changed changes how fast the player falls.

Make the following changes to “handle_movement”:

```
IF (my.movement_mode == 0) {
      my.gravity = 6;
      IF (my.move_x != 0 || my.move_y != 0) { //if we are moving
```

Make the following changes to “handle_gravity”:

```
my.force_z -= my.gravity * time;
my.force_z = max(-30,my.force_z);
```

Make the following changes to “player_action”:

```
ACTION player_action {
      my.gravity = 6;
      my.z_offset = 6;
```

Now the my.gravity variable will affect how fast the my.force_z variable decreases. Previously you just used the number 6, now you will be using a variable which you can change to work with your combos and animations to get them looking just right. At the beginning of “player_action” you have set my.gravity to 6, otherwise the default would be 0 and the player may get stuck in the air with no gravity to move it downwards. My.gravity also gets set to 6 in "handle_movement" when the player isn't attacking (my.movement_mode == 0), this is to set gravity back to normal when you aren't performing a combo as we only want to control it in combos.

Make the following changes to "handle_animation":

The attack_c if statement:

```
IF (combo_continue == 1) {
      my.jumping_mode = 10;
      my.force_z = 12;
      my.gravity = 4;
      combo_continue = 0;
      my.blendframe = attack_d;
      my.movement_mode = 1;
}
```

The attack_e if statement:

```
my.currentframe = attack_e;
IF (my.jumping_mode == 0 && my.animate >= 20 && my.animate < 60) {
      my.gravity = 3;
      my.jumping_mode = 10;
      my.force_z = 15;
```

The attack_f if statement:

```
my.currentframe = attack_f;
IF (my.jumping_mode == 0 && my.animate >= 40 && my.animate < 60) {
      my.gravity = 3;
      my.jumping_mode = 10;
      my.force_z = 8;
```

Now the gravity changes from attack to attack, making falling a little slower for attack_d and attack_e. Run your code and test it :)

Now it's time for airborne attacks :)

**Airborne Combos**

There are a few ways to attempt this. You could create a whole new set of animations for different combos in the air, or use the same attack animations with slightly different script, and just set my.gravity = 0. You could also set it up that you can only perform an airborne combo if the first attack strikes an enemy. You could also integrate a lock on system, which uses scan_entity to find the nearest enemy, and simply pan the player towards the enemy rather than the direction the keys are being pressed.

At the moment movement_mode is set at 1 for attacks, let's have movement_mode set at 2 for airborne attacks.

Make the following changes to "handle_movement":

```
IF (mouse_left == 1 && mouse_left_press == 0 && my.animblend >= stand) {
      mouse_left_press = 1;
      my.blendframe = attack_a;
      IF (my.jumping_mode == 1) { my.movement_mode = 2; } ELSE { my.movement_mode = 1; }
      combo_continue = 0;
}
```

If we press mouse left and are jumping by pressing the space bar (my.jumping_mode == 1), and not jumping from falling, set movement_mode to 2 (airborne attack) else set to 1 (normal attack).

Make the following changes to "handle_movement", we're just making sure the same type of rotation/movement is used for airborne combat as ground combat:

```
IF (my.movement_mode == 1 || my.movement_mode == 2) {
      temp.y = fsin((my.animate * 1.2) + 45,15 * time);
      my.move_x = fcos(my.pan,temp.y);
      my.move_y = fsin(my.pan,temp.y);
      temp.y = 0;
      IF (temp.x != -1000) { //if we need to rotate whilst attacking (the player is
pressing keys to rotate)
            temp.y = 1;
      }
}
```

Add the following to "handle_movement" after the "IF (my.movement_mode == 1)" if statement:

```
IF (my.movement_mode == 2) {
      my.jumping_mode = 0;
      IF (mouse_left == 0 && mouse_left_press == 1) { mouse_left_press = 0; }
      IF (mouse_left == 1 && mouse_left_press == 0 && my.animate >= 30 && combo_continue
== 0) { mouse_left_press = 1; combo_continue = 1; }
}
```

You could also integrate this into (my.movement_mode == 1) if you wanted, ie change to (my.movement_mode >= 1) and then add another if statement to set "my.jumping_mode = 0;" if (my.movement_mode == 2). I'm showing you this way instead for simplicity and perhaps more control over as it is in a separate if statement. This is doing the same as the (my.movement_mode == 1) if statement except the player doesn't need to fall, so my.jumping_mode is always set to 0.

Make the following changes to "handle_gravity":

```
my.force_z -= my.gravity * time;
my.force_z = max(-30,my.force_z);
IF (my.movement_mode == 2) { my.force_z = 0; }
```

If you are performing an airborne attack (my.movement_mode == 2) apply no force along the z axis "my.force_z = 0;"

Run your code, now as you jump in the air and attack you wont fall, this looks a bit funny but we'll refine it more. We can set it up so that the airborne attack only works once the first hit strikes an enemy, then the player will continue the combo until a hit misses or the combo completes, this will work like the combos in Kingdom Hearts.

There is a small issue, if you finish a combo attack the player will always blend to the walk or run animation, this is because the player is always moving whilst performing a combo (my.move_x != 0 || my.move_y != 0), we need to determine if the player is holding WSAD at the end of a combo, and based on that blend to the correct animation. To do this we will use my.moving, if set to 1 this means the player is holding down WSAD, if not this means the player isn't and you need to blend to the stand animation after a combo.

Make the following changes to handle movement:

```
IF (key_s == 1 && key_d == 1 && key_a == 0 && key_w == 0) { temp.x = camera.pan - 135; }
my.moving = 0;
IF (temp.x != -1000) {
      my.moving = 1;
      IF (key_shift == 1) { temp.y = 10 * time_step; } ELSE { temp.y = 15 * time_step; }
}
```

If we are moving (temp.x != -1000) set the my.moving variable to 1.

Now in handle_animation, change every occurrence of:

```
IF (my.move_x != 0 || my.move_y != 0) {
```

to

```
IF (my.moving == 1) {
```

Make sure you do not replace the line "IF (my.move_x != 0 || my.move_y != 0) {" in the "handle_movement" function, but only in "handle_animation".

Now when you finish an attack the player will blend to the correct animation.

# ENEMIES

This tutorial wont cover enemy ai and behavior, however in this next section you will simply set up a dummy enemy which your character can whack around, you'll also be able to lock on to your target.

Firstly let's add a new script, make the following changes to main.wdl:

```
include <player.wdl>;
include <enemy.wdl>;
include <animation.wdl>;
```

Create a new script named "enemy.wdl"

Time to use our awesome player pointer, this is an automatically predefined pointer, and can always be used by script, I am assuming you know what pointers are and how they work. Make the following changes to "player_action":

```
ACTION player_action {
      player = my;
      my.gravity = 6;
```

You also need to create a pointer for the player's weapon and for the enemy the player is attacking, add the following at the top of your script where your variables are:

```
entity* player_weapon;
entity* target_enemy;
```

Make the following changes to “attach_weapon”:

```
ACTION attach_weapon {
      player_weapon = my;
      my.passable = on;
```

Make the following changes to your variables:

```
var space_press = 0;
var mouse_left_press = 0;
var mouse_right_press = 0;
var combo_continue = 0;
var player_lock_on = 0;
var airborne_attack = 0;
```

Add the following function at the end of player.wdl:

```
FUNCTION handle_sword_collision {
      vec_for_vertex(temp.x,player_weapon,274); //sword base
      vec_for_vertex(temp2.x,player_weapon,54); //sword tip
      trace_mode = ignore_me+ignore_passable+use_box;
      result = trace(temp.x,temp2.x);
      IF (you != null) {
            IF (you.entity_type == 2 && you.hit_by_player == 0) {
                  IF (airborne_attack == 1 && my.animblend == attack_a) {
                        my.movement_mode = 2;
                  }
                  you.hit_by_player = 1;
                  IF (target_enemy == null) { target_enemy = you; player_lock_on = 1; }
            }
      }
}
```

Make the following changes to “handle_movement”:

```
IF (target_enemy == null) {
      IF (temp.y > 0) { rotate_entity(temp.x,30); }
} ELSE {
      vec_diff(temp2.x,target_enemy.x,my.x);
      vec_to_angle(temp2.pan,temp2.x);
      rotate_entity(temp2.pan,30);
}
```

Further down in the function:

```
IF (mouse_left == 0 && mouse_left_press == 1) { mouse_left_press = 0; }
IF (mouse_left == 1 && mouse_left_press == 0 && my.animblend >= stand) {
      mouse_left_press = 1;
      my.blendframe = attack_a;
      IF (my.jumping_mode == 1) {
            airborne_attack = 1;
      } ELSE {
            airborne_attack = 0;
      }
      my.movement_mode = 1;
      combo_continue = 0;
}
IF (mouse_right == 0 && mouse_right_press == 1) { mouse_right_press = 0; }
IF (mouse_right == 1 && mouse_right_press == 0) {
      IF (player_lock_on == 0) {
            c_scan(player.x,player.pan,vector(360,180,250),scan_ents | scan_limit |
ignore_me);
            IF (you != null) {
                  IF (you.entity_type == 2) { //make sure you've scanned an enemy
                        player_lock_on = 1;
                        target_enemy = you;
                  }
            }
      } ELSE {
            player_lock_on = 0;
            target_enemy = null;
      }
      mouse_right_press = 1;
}
```

Make the following changes at the end of the “handle_animation” function:

```
IF (my.animblend >= attack_a && my.animblend <= attack_f) { handle_sword_collision(); }
IF (my.animblend != blend && my.blendframe != nullframe) { my.animate2 = 0; my.animblend =
blend; }
```

Add the following code to your new enemy.wdl:

```
ACTION enemy_dummy {
      my.shadow = on;
      my.entity_type = 2;
      my.enable_scan = on;
      WHILE (1) {
            IF (my.hit_by_player == 1) {
                  my.move_x = player.move_x;
                  my.move_y = player.move_y;
                  my.move_z = player.move_z;
                  c_move(my,nullvector,my.move_x,use_aabb | ignore_passable | glide);
                  IF (player.animblend == blend || player.animblend < attack_a ||
my.animblend > attack_f) { my.hit_by_player = 0; }
            }
            IF (target_enemy == my && vec_dist(my.x,player.x) > 200) { target_enemy =
null; player_lock_on = 0; }
            wait(1);
      }
}
```

Now add the player.mdl as an entity to your WED level, but give the entity the action “enemy_dummy”, place as many as you like, place some in the air and some on the ground. Run your game. Now when you attack an enemy you will automatically lock onto that enemy. Alternatively you can right click and you will lock on to the nearest enemy within a 250 quant diameter scan cone. If you are locked onto an enemy press right click to break the lock on, or if you move out of the 200 quant range using vec_dist of the enemy the lock on will automatically break. Please note you have used 200 and 250, they are 2 different values, for some reason if you place the same value into c_scan as you do vec_dist, it seems they don't work exactly the same, using a higher value in c_scan than vec_dist seems to work better in this circumstance. When you perform an airborne attack you will only perform a combo in the air if the first strike hits an enemy. Let's take a look at the code:

```
FUNCTION handle_sword_collision {
      vec_for_vertex(temp.x,player_weapon,274); //sword base
      vec_for_vertex(temp2.x,player_weapon,54); //sword tip
      trace_mode = ignore_me+ignore_passable+use_box;
      result = trace(temp.x,temp2.x);
      IF (you != null) {
            IF (you.entity_type == 2 && you.hit_by_player == 0) {
                  IF (airborne_attack == 1 && my.animblend == attack_a) {
                        my.movement_mode = 2;
                  }
                  you.hit_by_player = 1;
                  IF (target_enemy == null) { target_enemy = you; player_lock_on = 1; }
            }
      }
}
```

Similar to finding the vertex of the player's hand and attaching the sword to it, we are now finding the vertex of the base of the sword and it's tip, then performing a trace instruction to determine if the sword as struck an enemy. Please note that you are using use_box in this trace instruction, this is using the player's hull as a trace rather than a straight line, I placed in use_box as I find it makes hitting the enemy easier, try removing it and you'll see that the sword would have to be 100% accurate and pass through the enemies hull to strike it, with use_box the sword doesn't have to be as accurate. Trace from the sword's base to it's tip “result = trace(temp.x,temp2.x)”.

If you have hit an entity (you != null). If the entity you have hit is an enemy and if it hasn't already been hit (you.entity_type == 2 && you.hit_by_player == 0).

```
IF (airborne_attack == 1 && my.animblend == attack_a) {
      my.movement_mode = 2;
}
```

Now that we are initiating attacks as ground attacks, whether you are jumping and attacking or not, you need to find a new way to initiate the airborne combo “my.movement_mode = 2;”, if you have hit an enemy (previous if statements), if you began this attack by jumping and are performing your first combo (airborne_attack == 1 && my.animblend == attack_a) initiate airborne combo.

```
you.hit_by_player = 1;
IF (target_enemy == null) { target_enemy = you; player_lock_on = 1; }
```

Set the enemy to being attacked “you.hit_by_player = 1;” if you aren't locked onto an enemy (target_enemy == null) set the enemy to lock on to “target_enemy = you;” turn n lock on mode “player_lock_on = 1;”

```
IF (target_enemy == null) {
      IF (temp.y > 0) { rotate_entity(temp.x,30); }
} ELSE {
      vec_diff(temp2.x,target_enemy.x,my.x);
      vec_to_angle(temp2.pan,temp2.x);
      rotate_entity(temp2.pan,30);
}
```

If you aren't locked onto an enemy (target_enemy == null) perform normal rotation IF (temp.y > 0) { rotate_entity(temp.x,30); }.

If you are locked onto an enemy, rotate towards that enemy, find the vector from the player to the enemy "vec_diff(temp2.x,target_enemy.x,my.x);" find the angle of this vector "vec_to_angle(temp2.pan,temp2.x);" rotate the player towards the enemy "rotate_entity(temp2.pan,30);"

```
IF (mouse_left == 1 && mouse_left_press == 0 && my.animblend >= stand) {
      mouse_left_press = 1;
      my.blendframe = attack_a;
      IF (my.jumping_mode == 1) {
            airborne_attack = 1;
      } ELSE {
            airborne_attack = 0;
      }
      my.movement_mode = 1;
      combo_continue = 0;
}
```

You had previously controlled airborne combos whether the player was jumping or not, now you are setting combo mode to always use ground based combos "my.movement_mode = 1;", you set "airborne_attack = 1;" if the player initiated the attack by jumping, this variable is then later used in "handle_sword_collision" to determine if the player has struck the enemy with an airborne blow:

```
IF (airborne_attack == 1 && my.animblend == attack_a) {
      my.movement_mode = 2;
}
```

In "handle_animation":

```
IF (my.animblend >= attack_a && my.animblend <= attack_f) { handle_sword_collision(); }
```

If the player is performing any attack animation (my.animblend >= attack_a && my.animblend <= attack_f) check to see if the sword has struck an enemy "handle_sword_collision();"

Let's look at the following addition to "handle_movement":

```
IF (mouse_right == 0 && mouse_right_press == 1) { mouse_right_press = 0; }
IF (mouse_right == 1 && mouse_right_press == 0) {
      IF (player_lock_on == 0) {
            c_scan(player.x,player.pan,vector(360,180,250),scan_ents | scan_limit |
ignore_me);
            IF (you != null) {
                  IF (you.entity_type == 2) { //make sure you've scanned an enemy
                        player_lock_on = 1;
                        target_enemy = you;
                  }
            }
      } ELSE {
            player_lock_on = 0;
            target_enemy = null;
      }
      mouse_right_press = 1;
}
```

If you have clicked the right mouse button (mouse_right == 1 && mouse_right_press == 0) if you aren't locked onto an enemy (player_lock_on == 0) scan for the nearest enemy, using a 250 quant sphere scan cone "c_scan(player.x,player.pan,vector(360,180,250),scan_ents | scan_limit | ignore_me);" If you scanned an entity with enable_scan turned on within that scan cone (you != null) if the entity is an enemy (you.entity_type == 2), turn on lock on mode "player_lock_on = 1;" set the enemy to the lock on target "target_enemy = you;".

If you have clicked the right mouse button and you are already locked onto an enemy "} ELSE {" turn off lock on mode "player_lock_on = 0;" set the lock on enemy pointer to null "target_enemy = null;". player_lock_on is used in the enemy code that you will look at now.

```
ACTION enemy_dummy {
      my.shadow = on;
      my.entity_type = 2;
      my.enable_scan = on;
```

By setting "my.entity_type = 2;" you now have a way of using trace and c_scan and to determine what type of entity you have scanned, say you wanted to add NPCs to your level, but you wanted a different type of interaction to occur when you scanned them, seeing as both your NPCs and enemies will have "enable_scan = on" c_scan will detect both enemies and NPCs, to determine if you have scanned an NPC or enemy, you could set "my.entity_type = 3;" for NPCs, then after performing the c_scan, if my.entity_type of the scanned entity is equal to 2 then you know you've scanned an enemy, if it is 3 you have scanned a NPC. Allow the entity to be detected by c_scan "my.enable_scan = on;"

```
WHILE (1) {
      IF (my.hit_by_player == 1) {
            my.move_x = player.move_x;
            my.move_y = player.move_y;
            my.move_z = player.move_z;
            c_move(my,nullvector,my.move_x,use_aabb | ignore_passable | glide);
```

If the enemy has been hit by the player's sword (my.hit_by_player == 1), move the enemy the same way the player is moving "my.move_x = player.move_x; my.move_y = player.move_y; my.move_z = player.move_z; c_move(my,nullvector,my.move_x,use_aabb | ignore_passable | glide);"

The enemy is caught up in a combo and will move along with the player as it attacks, however if the attack ends you want the enemy to stop moving with the player ie "my.hit_by_player = 0;"

```
IF (player.animblend == blend || player.animblend < attack_a || my.animblend > attack_f) {
}
```

If the player's animation is blending (player.animblend == blend) or if the player's current animation is anything but an attack animation (player.animblend < attack_a || my.animblend > attack_f) stop the enemy moving with the player “my.hit_by_player = 0;”. Remember the player always blends from each attack to the next, so there will always be at least 1 frame between attacks where “player.animblend == blend”, this way if the player chooses not to continue a combo the enemy can stop moving with the player.

```
IF (target_enemy == my && vec_dist(my.x,player.x) > 200) { target_enemy = null;
player_lock_on = 0; }
```

This code turns off the lock on system. If the player is locked onto us (target_enemy == my) and if the distance from the player to us is greater than 200 quants (vec_dist(my.x,player.x) > 200) turn off the lock on “target_enemy = null; player_lock_on = 0;”. At the moment to only way to break a lock on with an enemy is to walk out of the 200 quant range or right click.

That's it. You've completed this tutorial! Congratulations. There is so much more to add and to tweak if you want to get your movement code to the state you want it in, it's an adventure I hope you enjoy.

# EXTRAS

Once you understand the frame work and basic elements to movement programming NPCs, enemies and other moving objects operate around the same principles. An enemy is exactly the same as the player except that it's movement is controlled differently. Enemies that are stationary and don't move are a lot easier to program than enemies that move and need to avoid collision or use path finding techniques. A most basic enemy would run through it's idle animation, and when triggered by an event say the player approaches, it then begins to process attack animations. The attack animations would use the same process as the player combat, you play the animation once and trigger events, create particle effects and perform collision based on where the animation is at. At the end of each attack the enemy can then decide what to do next using if statements.

If you can grasp and understand the concepts of the player movement above then I assume you are either in the place to begin implementing enemies or learning how to. I found that once I discovered how I could control objects fully and smoothly through c-script that I was flooded with ideas and possibilities of what I could create.

**Nexus and Virtual Memory**

Firstly, things I mention about nexus etc may not be 100% accurate, I am merely writing what I understand it to be. Some things I have found from the 3DGS manual and forums.

The Nexus is how much virtual memory is assigned when a level is loaded. I heard the advised amount is 200mb as that is an amount which most PCs, even older ones, are compatible with. Making it higher is fine (I think) it simply means that older pcs may not be compatible with it.

Whenever you add a new model or map entity to your level, it has to be loaded into virtual memory. In the debug panel (F11) you can see how much nexus your current level is using. I have found on average, a 200 frames, 512, 2000 poly model roughly uses 10 – 20mb in nexus, a character with 500 frames uses 30mb or so, vertex animations. Now with each new model you add that uses an extra 10-20mb in virtual memory, this means increased loading time, estimating 1-3 seconds or so. When you add a new model to your level keep and eye on the nexus in the debug panel, make sure it's not some really high number as this means the level will take a long time to load. And take note of how much memory new models use. I have found a 10mb med file on the hard drive (with 1000 vertex animation frames) uses 40 – 50mb or so in virtual memory, adding the model added 4 seconds of loading time, I went through and deleted every second frame from the animation, and deleted animations that weren't needed, I got the file down to 600 frames and it then used 30mb in nexus. This made a huge difference in loading time after I did this to many models. Bone animations use less virtual memory than vertex animations, use them wherever possible.

If you look at the debug panel in the level you created for this tutorial, the nexus is around 90mb or so, this is a lot, and most of it is used through the 3500 poly 500 frame vertex animation model.