DIABLO STYLE WORKSHOP for GameStudio A5

for A5 engine 5.15

By Brandon R. Batie (email: acidcrow@acidcrewsoftware.com)
AcidCrew Software (website: http://www.acidcrewsoftware.com) /
Conitec April 2002

Author: Brandon R. Batie

Enhancement: Tobias Runde

Contents

Foreword

This workshop is designed to show you how to create a basic "Diablo" style game in Gamestudio. This workshop assumes you have a knowledge of the basics of WDL and using the World Editor. You will want to have the latest version of Gamestudio and the editors.

You will also want to create a new directory in your Gstudio folder for your working directory called "Diablo" and copy the pguard3.mdl, mace.mdl, shield.mdl and and hero.mdl and the diablo.wmp, diablo.wmb and diablo.wdl into that directory. You can open the diablo demo level and see how the code works before you start this workshop. You will also need to edit the camera.wdl file (note: always back up your wdl files, or put them in them working directory and edit them there). The demo level uses the standard wad file.

How the code will work

When the player clicks on a spot on the map we will move the player to that position. If the player right clicks while move the player will stop. If they right click while not moving we will cast a spell to the place clicked on the map. The player will also have a run mode that can be toggled on and off with a key press.

SETTING UP WORKSPACE

Create a new directory in your Gstudio directory called "Diablo Workshop". This is where all the files needed will be kept and where you will create new files for the game.

Copy the files included in the zip file into this directory, it should at least have the following files:

Hero.mdl
Mace.mdl
Pguard3.mdl
Cube.mdl
Camera.wdl
My_code.wdl

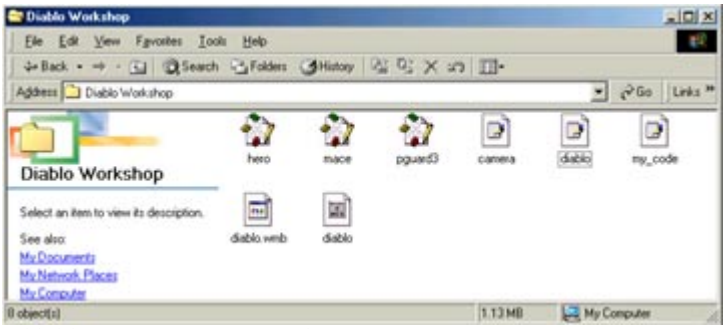

Abbildung 1: Create a new directory

THE TEST LEVEL – USING PRE-MADE CODE

For our test level we don't need anything fancy at this time. We can start by creating a new file project:

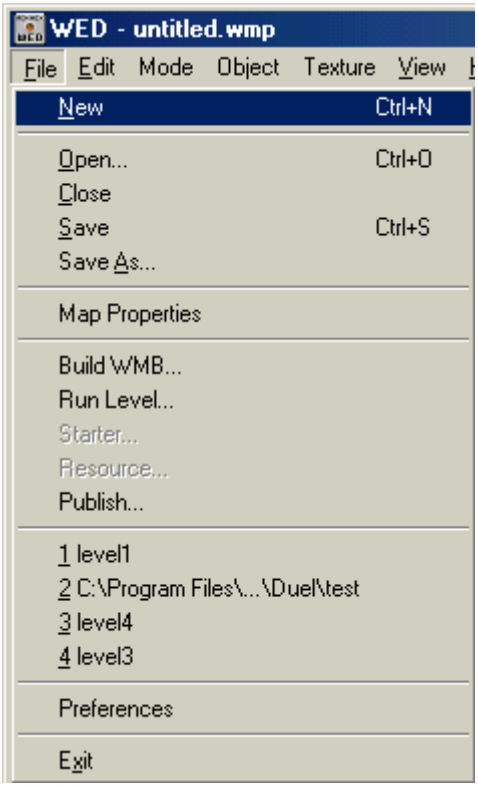

Abbildung 2: Create a new project

Add a large cube and scaling it up to a reasonable size to walk around in.

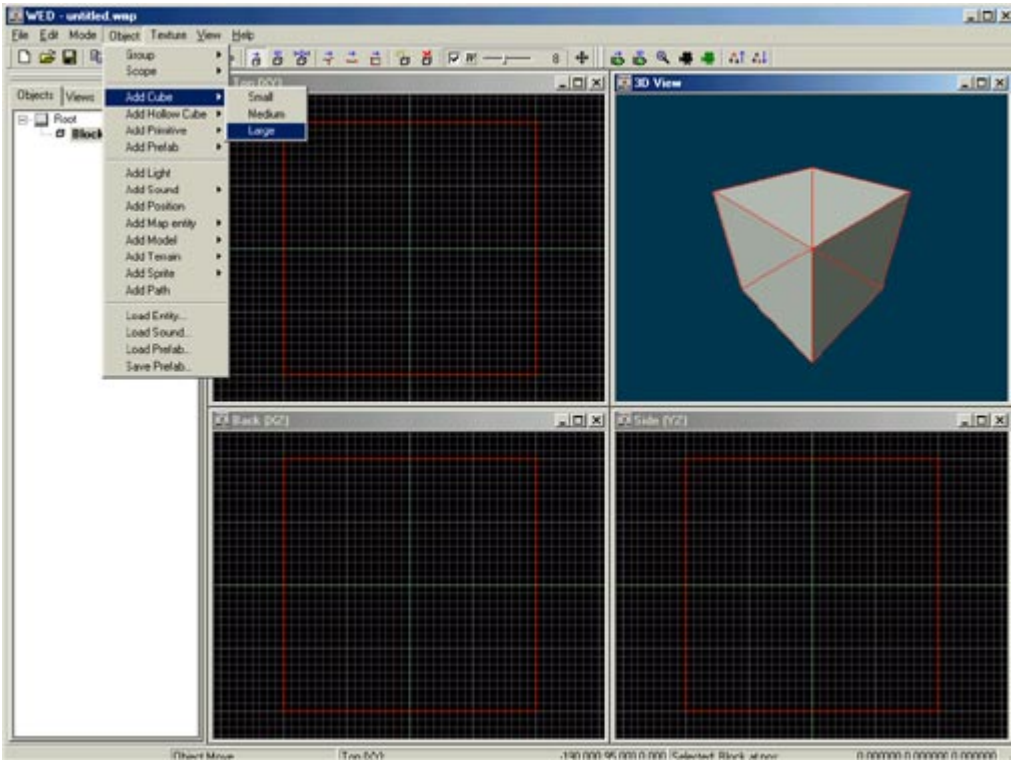

Abbildung 3Add a cube

Texture it with the grass texture and then we can then hollow the cube, scope down and delete the top part of the cube by selecting it (it will turn red) and pressing the delete key.

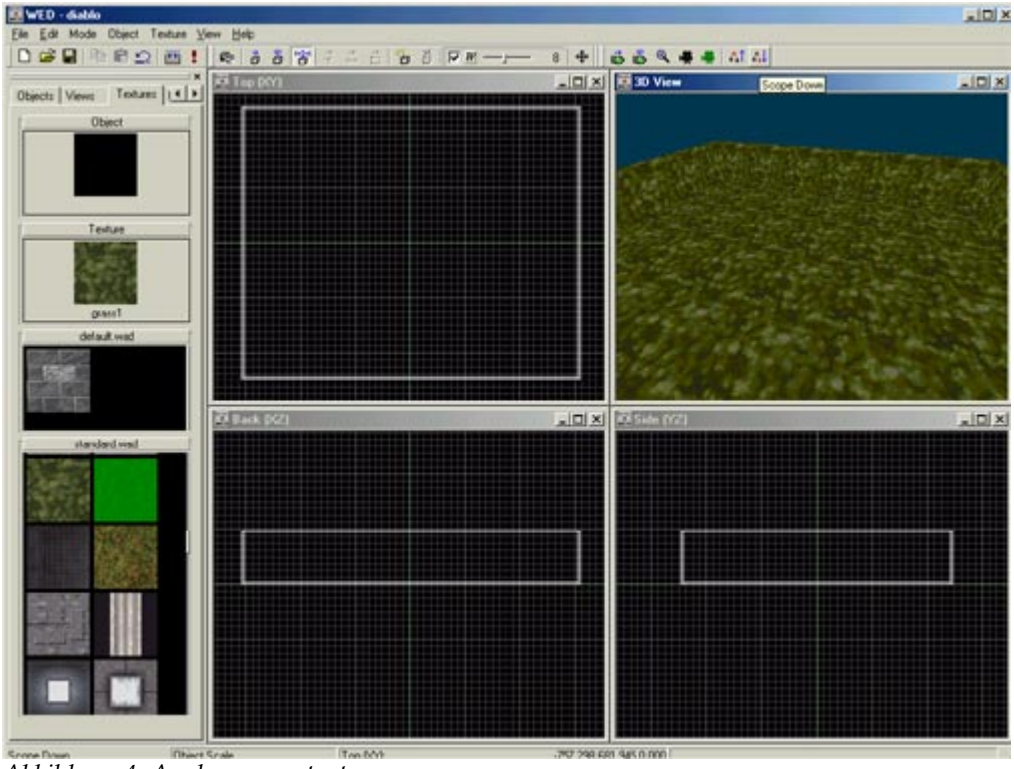

Abbildung 4: Apply a grass texture

This will give us a map with walls the player cannot walk beyond. We can now save our level as "Diablo", add our level script by clicking on File, Map Properties, then click the white icon for a new script and a new wdl file will be created in our working directory called diablo.wdl.

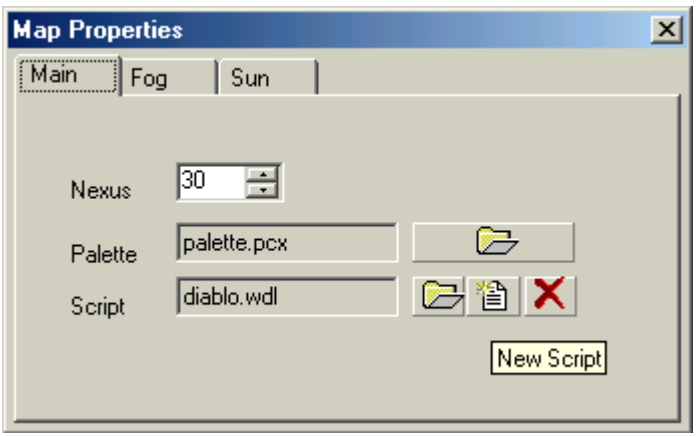

Abbildung 5: Choose a wdl file

We can now add the model we want to use as the player to the level.

Object…Add Model
If it is located in your working directory.
If you did not include it in your working directory you will need to click on Object…Load Entity and find the directory you saved the model in.

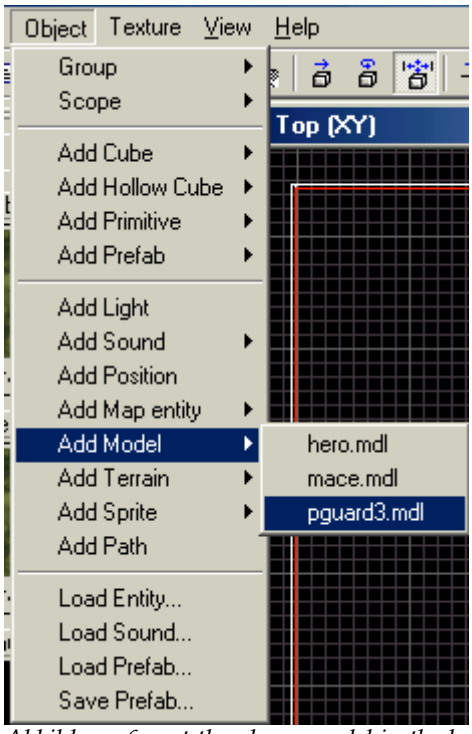

Abbildung 6: put the player model in the level

We won't need to create a sky box around our level since at the edges we would be able to see the ground below us and that wouldn't look right. Instead, what we will do is make it so it displays a solid black background.

We can also add some blocks for obstacles in various places on our map, but this is up to you.

We will now need to inlcude the my_code.wdl file in the diablo.wdl we just created. To do this we open the diablo.wdl and find the includes located near the top of the file. They should look something like this:

```
////////////////////////////////////////////////////////////////////////
///////
// The INCLUDE keyword can be used to include further WDL files,
// like those in the TEMPLATE subdirectory, with prefabricated actions
include <movement.wdl>;
include <messages.wdl>;
include <menu.wdl>;       // must be inserted before doors and weapons
include <particle.wdl>; // remove when you need no particles
include <doors.wdl>;       // remove when you need no doors
include <actors.wdl>;   // remove when you need no actors
include <weapons.wdl>;  // remove when you need no weapons
include <war.wdl>;      // remove when you need no fighting
//include <venture.wdl>;    // include when doing an adventure
include <lflare.wdl>;   // remove when you need no lens flares
```

Under the last one we will want to include our my_code.wd:

```
include <my_code.wdl>;
```

Now we can save our project and reopen it to access the newer actions from the my_code.wdl file.

Once open select the pguard3.mdl we added earlier, right click and click on Properties. This will open the Object Properties dialog box. Once open click the Behavior tab at the top.

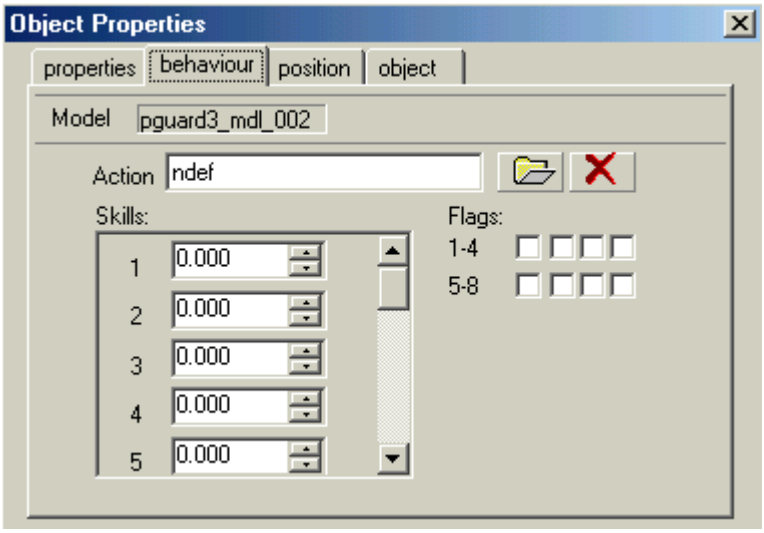

Abbildung 7give the entity an action

Click the Yellow folder icon to open the Chose Action dialog.

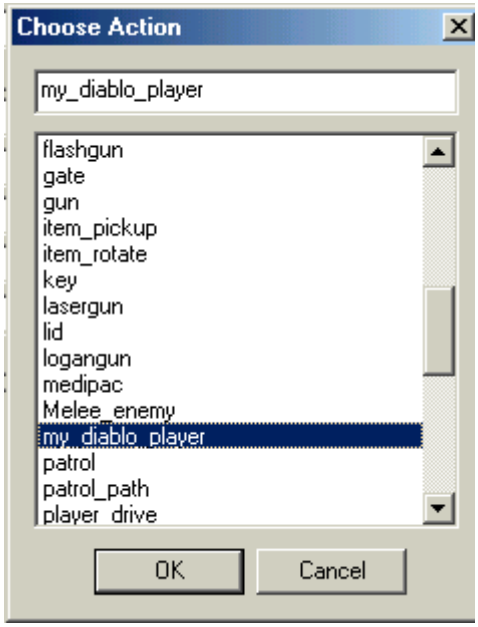

Abbildung 8: Choose an action

Scroll down if needed and find the my_diablo_player action, select it and then click ok.

Using the same steps as above you can also add the hero.mdl and assign the action of one of the template enemies robot1, robot2, robot3 and actor_walk_fight.

The player code in the my_code.wdl will automatically create the mace as the players weapon with the needed attributes.

We can now save and build our map then test it. You should now be able to click to a spot on the map and the player will walk to that position. Right clicking on the map should cast a spell to that position. If you hold down the shift key and left click the player should swing the mace to attack.

How to code the script form scratch

This part shows how to code the script from the scratch. For this the script will divide in several parts. First we'll code a special view, for the Diablo Style. Second, the player must move. To attack your enemies, you learn to combat close and use spells. When a enemy is dead, he drops an item for your player. But now we begin to code the view.

Our View

Open the diablo.wdl file, and in Function Main() near the end you will see the line:

```
// call further functions here...
```

After that line we can add our code to set the background color for the level:

```
bg_color = 5;
```

This will gives us the black background and also stop the edges that we can see over from getting the 'streaking' effect.

First, since this is a top down view game we will want to define our camera code.

If we open the camera.wdl file near the top we will find the lines:

```
var orbit_camera_pan = 180;   // pan around center point
var orbit_camera_dist = 150;  // distance from center point
var orbit_camera_zOff = 5;     // distance up
```

By simply changing these lines we have a near perfect diablo style camera position.

```
var orbit_camera_pan = 180;   // pan around center point
var orbit_camera_dist = 250;  // distance from center point
var orbit_camera_zOff = 450;      // distance up
```

But, to make it start in that mode we will have to add some code to our Main function once again. Just below where we defined the *bg_color* we will add:

```
person_3rd  = 2;
```

This will activate our camera code on startup.

Next we will want to define our player and their movement code. For this we will need three Functions and one Action. Personally, I like to define the combined functions and actions then add the code to them. But, to simplify this workshop I will mention the ones we need and we will code them separately. We will first need the following:

1. Action my_diablo_player – this will be the player definition.
2. Function change_run – this will toggle the run mode on and off.
3. Function get_target – this will get the target on the map that the player will move to.
4. Function my_diablo_move – this is the code that will actually move the player to the target position.

Run Mode

We will now want to create a new wdl file and save it as my_code.wdl in our working directory. This will need to be included in our diablo.wdl file.

The first thing we will want to do is to define our player skills with easy to use names.

```
DEFINE _IS_WALKING, SKILL33;
DEFINE _RUN_MODE, SKILL34;
```

We will use the player's skill 33 to know if they are walking or not and call it _is_walking. We will also define the player's skill 34 as _run_mode. We can now access them as:

```
Player._is_walking
```

and

```
Player._run_mode
```

We will first do the run_mode function. So, we define the function:

```
Function change_run()
{

}
```

Since we have already defined the players skill34 as _run_mode we can use that in our code:

```
Function change_run() //The function to toggle run mode off and on
{
   player._run_mode = (player._run_mode == off);
}
```

Now we need a way to let the player toggle the run mode. We will do this with the R key on the keyboard.

```
On_r change_run; //when the R key is pressed it will run the above function and toggle the run //mode
```

For good coding practice and to make code easier to find later key press definitions should be at the end of the file.

Next we will define where the player wants to go to. For this we use our function get_target. When the player clicks on the map this is where they will move to. So we can now define our function:

```
Function get_target()
{

}
```

Now, to get where the player clicks Conitec was kind enough to include a default code for us called *mouse_to_level*. This will convert the mouse position to our world position vector. So we will first call that.

```
Function get_target()
{
   mouse_to_level();
}
```

Mouse_to_level will set the mouse position to a pre-defined vector called target. Since the templates in other functions use target also we need to make sure we don't lose this vector due to some other function also using it. We will need to define two variable arrays for this function. My_target and move_position. We use two variables here so when the player clicks on a new location it doesn't interfere with the position that is already set until we assign it with the command,

```
Var my_target[3]; //this defines the target variable, the [3] gives us an array of three for
                  //our x, y and z positions.
Var move_position[3]; //this defines the move position variable the [3] gives us an array of three
                      //for our x, y and z positions.
```

You can use the single variable for this function, but since they interfere with each other just a bit it is harder to check if the player is at the target position later in the code.

You will want to define these and other variables near the top of the script, or you could get an error from trying to access and variable that hasn't be defined.

To assign the make my_target the same as the target vector from the *mouse_to_level* function we use *vec_set(vec1,vec2);* with this the vector called vec1 will be the same as the vector called vec2. So we will add that inside our function below our mouse_to_level call.

```
Function get_target()
{
   mouse_to_level();
   vec_set(my_target,target); //my_target becomes the same as target so we don't lose our vector
}
```

Now that we know where we want the player to move to we first need to turn the player to that position. Our first command in this will be *vec_sub(vec1,vec2);* this will subtract vec2 from vec1.

```
Function get_target()
{
   mouse_to_level();
   vec_set(my_target,target); //my_target becomes the same as target so we don't lose our vector
   vec_set(move_position,target); //move position becomes the same as target so we don't lose our
                                  //vector
   vec_sub(move_position,player.x); //we subtract the player position from the move position
}
```

We can now turn the player to the my_target vector by using *vec_to_angle(vec1,vec2);* this will set the vec1 angle to the vector of vec2.

```
Function get_target()
{
   mouse_to_level();
   vec_set(my_target,target); //my_target becomes the same as target so we don't lose our vector
   vec_set(move_position,target); //move position becomes the same as target so we don't lose our
                                 //vector
   vec_sub(move_position,player.x); //we subtract the player position from the move position
   vec_to_angle(player.pan,move_position); //set the players angle to the my_target position
   player.tilt = 0; // Here we are making sure the player is not tilted, otherwise you player may
                   //be in a weird position
}
```

Now that we have our target position that we want to move the player to we need the player to know that they have a target. We can now use our player._is_walking to let the player entity know it has a target and should move.

```
Function get_target()
{
   mouse_to_level();
   vec_set(my_target,target); //my_target becomes the same as target so we don't lose our vector
   vec_set(move_position,target); //move position becomes the same as target so we don't lose our
                                  //vector
   vec_sub(move_position,player.x); //we subtract the player position from the move position
   vec_to_angle(player.pan,move_position); //set the players angle to the my_target position
   player.tilt = 0; // Here we are making sure the player is not tilted, otherwise you player may
                   // be in a weird position
   player._is_walking = 1; //let the player entity know it has a target position
}
```

We will also want the player to be able to attack without moving. To do this we will check to see if the shift key is pressed, if so we will only attack and not move.

```
if(key_shift == 1) //the shift key is pressed (1)
{
   player._is_walking = 0; //set player mode to not moving
   weapon_fire(); //attack with the current weapon
   return; //leave the function
}
```

We will add this to the existing code after we rotate to the position clicked, since this should be the direction we want to attack in.

```
Function get_target()
{
   mouse_to_level();
   vec_set(my_target,target); //my_target becomes the same as target so we don't lose our vector
   vec_set(move_position,target); //move position becomes the same as target so we don't lose our
                                  //vector
   vec_sub(move_position,player.x); //we subtract the player position from the move position
   vec_to_angle(player.pan,move_position); //set the players angle to the my_target position
   player.tilt = 0; // Here we are making sure the player is not tilted, otherwise you player may
                   //be in a weird position
   if(key_shift == 1) //the shift key is pressed (1)
   {
      player._is_walking = 0; //set player mode to not moving
      weapon_fire(); //attack with the current weapon
      return; //leave the function
   }
   player.skill48 = 1; //let the player entity know it has a target position
}
```

Next we need a way to call the function, of course this is done by left clicking the mouse button on the level. To do this we will assign the get_target function to the left mouse click:

```
On_mouse_left get_target; //when the left mouse button is clicked call the get_target function
```

Now, we have our run mode and we have our target position so now we need to actually move the player entity to that position. This will be the my_diablo_move function so we need to define that.

```
Function my_diablo_move()
{

}
```

We will want this code to constantly check for when the player should move. For this we will use a *while(1)* command. *While(1)* means that it will always perform the code inside the *while(1)*. So we will add that to the function.

```
Function my_diablo_move()
{
   while(1) // always run this code
   {

      wait(1); // we always need to add a wait() command in a while loop
               // to avoid an infinite loop error
   }
}
```

Since we have set the _is_walking to 1 when we have a target we need to check that in our loop.

```
Function my_diablo_move()
{
   while(1) // always run this code
   {
      if(player._is_walking == 1) //the player has a target to move to
      {

      }
      wait(1); // we always need to add a wait() command in a while loop
               // to avoid an infinite loop error
   }
}
```

Now that we know if the player has a target we need to know if we are close enough to the target to stop and if not we want to move there. For this we will compare the player vector and target vector with *vec_dist(vec1,vec2);* this will get the distance between the two vectors.

```
Function my_diablo_move()
{
   while(1) // always run this code
   {
      if(player._is_walking == 1) //the player has a target to move to
      {
         if (vec_dist(player.x,my_target.x) > 35) //is the distance too great
         {

         }
      }
      wait(1); //we need to wait for the next frame before we run the loop again
   }
}
```

You may noticed I check if the distance between the two is greater than 35. The X position of the model can vary depending on who made the model since the X position goes by the models origin. The model I am using is the pgurad3.mdl from the Contiec download page in the Geo-Metricks teaser pack. I usually start with half the models size in WED and see where and if it stops, normally you will want it just above half way. If it keeps going raise the number until it stops where you want it. If it stops before it gets to the position you clicked lower the number.

Now that we know the distance is too great and we need to move the player we will want to see if we should run there or walk there. To do this we need to see if we are in run mode or not.

```
Function my_diablo_move()
{
   while(1) // always run this code
   {
      if(player._is_walking == 1) //the player has a target to move to
      {
         if (vec_dist(player.x,my_target.x) > 35) //is the distance too great
         {
            if(player._run_mode == 1) //if the player is in run mode
            {

            }
            else //if the player is not in run mode
            {

            }
```

```
            }
         }
         wait(1); // we need to wait for the next frame before we run the loop again
      }
   }
```

Now we need to know how fast to move the player. In the template code this is defined by *my._force*. 1 is the default force. Since we haven't changed it we know the force is 1, we can set our run speed to 2.5 by taking the *my._force* and multiplying it by 2.5.

```
Function my_diablo_move()
{
   while(1) // always run this code
   {
      if(player._is_walking == 1) //the player has a target to move to
      {
         if (vec_dist(player.x,my_target.x) > 35) //is the distance too great
         {
            if(player._run_mode == 1) //if the player is in run mode
            {
               force = my._force * 2.5; //Force is used by the
                                        // template code to move the entity at that
                                        //speed, so we will multiply _force by 2.5 and
                                        //assign that to force
            }
            else //if the player is not in run mode
            {
               force = my._force; // we are not in run mode so we don't change the force
            }
         }
      }
      wait(1); // we need to wait for the next frame before we run the loop again
   }
}
```

Now all we need to do is move the player entity. We can use a function pre-defined for us in the templates called *move_gravity();* this will move the entity in the direction it is facing and perform collision detection and gravity force on the entity so if we encounter something it will react to it.

```
Function my_diablo_move()
{
   while(1) // always run this code
   {
      if(player._is_walking == 1) //the player has a target to move to
      {
         if(vec_dist(player.x,my_target.x) > 35) //is the distance too great
         {
            if(player._run_mode  == 1) //if the player is in run mode
            {
               force = my._force * 2.5; //Force is used by the
                                        // template code to move the entity at that
                                        //speed, so we will multiply _force by 2.5 and
                                        //assign that to force
            }
            else //if the player is not in run mode
            {
               force = my._force; // we are not in run mode so we don't change the force
            }
            move_gravity();

         }
      }
```

```
      wait(1); // we need to wait for the next frame before we run the loop again
   }
}
```

One thing you may notice is that the player entity will move until it reaches the target position. Due to various in game factors the player may want to stop the entity without setting a new target position. This can be done with a key press or a mouse press. Since most Diablo style games if the player is moving and they left click again it sets that as a new target we will want to keep it that way. I prefer to have the player stop on a right click, but I also like to have the right click to cast spells. So to get by this we will later create a function that does both.

Now that we have all of our movement code we need to have a player action to assign to our entity.

```
Action my_diablo_player
{

}
```

Inside this we will first want to call the default *player_walk_fight()* action so our player can take damage, be detected by enemies and set all the default properties.

```
Action my_diablo_player
{
   player_walk_fight();
}
```

We then need to call the movement code we wrote to start the functions loop.

```
Action my_diablo_player
{
   player_walk_fight();
   my_diablo_move();
}
```

We can now go back into our level (save and reload it so we can access the newest changes) and assign our player model the my_diablo_player action and test the level. We do this by selecting our model, right clicking and clicking on Properties. Then click the Behavior tab, then the yellow folder icon to open the chose action dialog and select my_diablo_player.

If you followed the code word for word (as in cut and paste) and used the models that came with it you should have a working diablo style movement.

Close Combat

Many people have asked on the forum for close combat code. While it is really very easy to do. You can define close combat by using the standard weapons code and simply modifying the bullet speed of the weapon, placing the model in the level and assigning the action to it.

Here I define my guard's mace taken from the default startgun1 and modified.

```
function startmace()
{
   my._ammotype = 0.0;
   my._weaponnumber = 1;
```

```
    my._bulletspeed = 30.05;
    my._firetime = 7;
    my._firemode = damage_shoot+hit_hole+0.30;
    gun_givePlayer();
}
```

The function stats are as follows:

_ammotype – sets ammo number and amount in gun. 0 means no ammo needed which is what we want since our player will always have the mace.

_weaponnumber – this is the number the player can press to select this weapon, this will allow you to add more weapons just like in an FPS type game.

_bulletspeed – this is an important one to use, since the first number defines how far to 'shoot'. We keep this at a low number to give the illusion of hitting with the mace. The second number is the recoil for the weapon and mainly affects the FPS mode.

_firetime – this is how long it takes before the player can attack with this weapon. It also defines how long the attack animation will play. You will have to adjust this according to the model animation.

-firemode – also kind of important. Here we set the damage_shoot and hit_hole, this will cause the weapon to create a hole where it impact on the level. Since the top of the mace is round this works out well, for an axe or sword you would want to edit the blthole.pcx file for a slash instead of a round hole.

Gun_giveplayer() – this is a template function that will assign this weapon to the player.

You may notice that the weapon code is a function and not an action, meaning we can't assign it to a model in WED. Instead we will write some additional code that will create the mace model inside the level and attach it to the player model.

To do this we will create another function called my_mace.

```
Function my_mace()
{

}
```

Inside this function we want to call the *attach_entity()* function. This is a default template function that will attach it to the entity that calls the function.

```
Function my_mace()
{
    attach_entity();
}
```

Next we will call the startmace() function to set its properties.

```
Function my_mace()
{
    attach_entity();
    startmace();
}
```

To get the player the model we will need to call it from the player action with the create command.

Ent_create("mace.mdl",player.x,my_mace); - this will create the mace model at the players position and assign it the my mace action that will attach it to the player and set its properties.

We will first want to define the string we will use for *ent_create* :

```
string mace_mdl = <mace.mdl>;
```

We can then reference the mace file by calling <mace.mdl>.

```
Action my_diablo_player
{
   player_walk_fight();
   my_diablo_move();
   ent_create(mace_mdl,player.x,my_mace);
}
```

We should now be able to test our level. Inside the level we should be able to walk around and hit walls (and anything else if you happened to add it to the level).

Spells

A spell casting ability can really improve the game play. We will start off with a really basic spell with no effect other than a flash of light. This will give you an idea of how to create spells and add various effects to them.

First we will define our function, cast_spell.

```
Function cast_spell()
{

}
```

Since we will be using the right click to cast a spell we need to get the position that was clicked. Since we have the code we used to get the target position of the player we can copy part of that and use it in our spell code, but we will want to define a new array variable for it so we don't interfere with the player movement.

```
Var spell_target[3];
```

For this function we will only need the single variable.

```
Function cast_spell()
{
   mouse_to_level(); //gets mouse to screen vector
   vec_set (spell_target, target);//set spell target to the target vector
   vec_sub(spell_target,player.x); //subtract the player pos for the spell target
   vec_to_angle(player.pan,spell_target); //turns player to target vector
   player.tilt = 0; //make sure the player is not tilted
}
```

We now have the position that we want our spell to be cast at. We now need to make it do some damage to anything there. This can easily be done by creating a entity at the position and applying some explode damage to it.

We need a function for the entity.

```
Function spell_event()
{
   damage = 50;
   range = 100;
   exclusive_entity;
   my.event = null;
   my.passable =  on;
   my.invisible = on;
   my.ambient = 100;
   my.red = light_flash.RED;
   my.green = light_flash.GREEN;
   my.blue = light_flash.BLUE;
   my.lightrange = 64;
   wait(1);

   temp.pan = 360;
   temp.tilt = 360;
   temp.z = range;
   indicator = _explode; // must always be set before scanning
   scan(my.pos,my_angle,temp);
   waitt(8);  //wait for ½ a second to make sure we get the flash effect

   remove(me);
}
```

Many of the properties of the spell event won't effect how it works, only how it looks when cast. Here is a break down of what each one does.

Damage – sets how much damage is done by the spell, the damage will decrease with the distance entities are from its center.

Range – the range the spell will affect.

Exclusive_entity – stops other actions

My.event – you can use this to assign effects and other functions to the entity (using Gaehhs fire pack try assigning a fire function to it…)

My.passable – this makes the entity passable, you can also set this to off if you want the entity to stop enemies for shot periods of time…

My.invisible – sets the entity to invisible so we can't see the model

My.ambient – sets the ambient of the entity.

My.red – sets the red color of the light the entity displays.

My.green – sets the green color of the light the entity displays.

My.blue – sets the blue color of the light the entity displays.

My.lightrange – the range of the flash for the spell

Temp, **pan**, **tilt** and **z** define the spherical range of the damage that will be done.

Indicator – tells the templates that this is an explosion to vary damage it does.

Scan – performs a scan function to apply the damage to nearby entities.

Remove(me) – the spell is over and we can remove our entity.

NOTE: You can also include calls to create special effects (if you include the Fire Pack's, fire.wdl, try calling *my_explode();* at the first part of the script.

We now need to create a simple action for our spell entity called spell_entity.

```
action spell_entity
{
   my.invisible = on; //make sure we can't see the entity
   my.event = spell_event(); //call the spell_event function
}
```

Then we add the create code to the cast_spell function:

First we define the string for the cube we will use:

```
string cube_mdl = <cube.mdl>;
ent_create(cube_mdl,spell_target,spell_event);
```

Added to existing code:

```
Function cast_spell()
{
   mouse_to_level(); //gets mouse to screen vector
   vec_set (spell_target, target);//set spell target to the target vector
   vec_sub(spell_target,player.x); //subtract the player pos for the spell target
   vec_to_angle(player.pan,spell_target); //turns player to target vector
   player.tilt = 0; //make sure the player is not tilted
   ent_create(cube_mdl,spell_target,spell_event);
}
```

If you remember from earlier we wanted the right click to also stop the play from moving. We have two choices now:
1. We can stop the player and cast a spell with one click.
2. We can stop the player and make the right click again to cast the spell.

I prefer the second option. So in the movement code we want to check for the right mouse click just before the *move_gravity()* call.

```
if(mouse_right == 1){wait(1);player._is_walking = 0;break;}
```

Adding that to the existing code:

```
Function my_diablo_move()
{
   while(1) // always run this code
   {
      if(player._is_walking == 1) //the player has a target to move to
      {
         if(vec_dist(player.x,my_target.x) > 35) //is the distance too great
```

```
        {
            if(player._run_mode == 1) //if the player is in run mode
            {
            force = my._force * 2.5; //Force is used by the
                                     // template code to move the entity at that
                                     //speed, so we will multiply _force by 2.5 and
                                     //assign that to force
            }
            else //if the player is not in run mode
            {
               force = my._force; // we are not in run mode so we don't change the force
            }
            if(mouse_right == 1)
            {
               wait(1);
               player._is_walking = 0;
               break;
            }
            move_gravity();
         }
      }
      wait(1); // we need to wait for the next frame before we run the loop again
   }
}
```

We will also want to check the cast_spell code so that if _is_walking is greater than 0 we don't do anything.

```
If(player._is_walking > 0)  {return;} //player is moving so we exit the function
```

Added to the existing code:

```
Function cast_spell()
{
   If(player._is_walking > 0)  {return;} //player is moving so we exit the function

   mouse_to_level(); //gets mouse to screen vector
   vec_set (spell_target, target);//set spell target to the target vector
   vec_sub(spell_target,player.x); //subtract the player pos for the spell target
   vec_to_angle(player.pan,spell_target); //turns player to target vector
   player.tilt = 0; //make sure the player is not tilted
   ent_create(cube_mdl,spell_target,spell_event); //create the cube with the spell_event function
}
```

Now if the player is moving they will stop, if the player is not moving they will automatically cast the spell.

We can now assign the right mouse click the cast_spell function.

```
On_mouse_right = cast_spell;
```

To make spell casting more fun we could also add a mana variable. When a spell is cast it will subtract from the mana, giving the player limited spell ability. To do this we would need to define a new variable my_mana and set its starting number.

```
Var my_mana = 100;
```

Now, when the player casts a spell we can subtract from the amount of mana they have.

```
My.mana -= 10; // each time a spell is cast it costs 10 mana points.
```

You can add this in the cast_spell function.

```
Function cast_spell()
{
   If(player._is_walking > 0)  {return;} //player is moving so we exit the function

   mouse_to_level(); //gets mouse to screen vector
   vec_set (spell_target, target);//set spell target to the target vector
   vec_sub(spell_target,player.x); //subtract the player pos for the spell target
   vec_to_angle(player.pan,spell_target); //turns player to target vector
   player.tilt = 0; //make sure the player is not tilted
   ent_create(cube_mdl,spell_target,spell_event);
   My_mana -= 10; // each time a spell is cast it costs 10 mana points.
}
```

We will also want the player to know how much mana he has left. We can define some digits to do this with just a few lines of code. We will put it in the top left corner.

```
PANEL player_mana //defines the panel with the name player_mana
{
   layer = 1; //sets the layer to 1
   pos_x 0; //sets the X position of the panel at 0
   pos_y 0; // sets the Y position of the panel at 0
   digits 0,0,7,digit_font,1,my_mana;//digitssets the X position,
                                     //Y position, length, font bitmap used, variable to track
   flags refresh,visible; //set the flag to refresh and the panel to //visible
}
```

Enemies dropping Items when killed

Most of these style games have items that are dropped when an enemy dies. This can easily be done by adding one line in the war.wdl template code. Open the war.wdl file, click on search and type in *FUNCTION STATE_DIE*, this will take you to the function that tells when an entity is out of health. At the end of the function will be the line: *my.passable = on; //body remains*. Between this line and the } you can use a create function to create an object for the player like health.

```
string health_mdl = <health.mdl>;
ent_create(health_mdl,my.pos,medipac);  // will create a medi pack to increase
                                        // the players health at the entities location.
```

Final Thoughts

This is just a basic code for a diablo style game. But, using this workshop and the concepts in it there is a wide range of options for you to include. Programming can be tough and taxing work in some cases and no good game is built quickly, they take planning and time.

Hopefully all the readers of this will understand not only how to create the Diablo style games, but game design and programming in WDL in general.

Abbildung 9: The Game

Index