Hello! In this part of documentation you will find out how to create logics for your own level.
version: final, author: Pejti
Special thanks: Zax37, Kubus_PL

Table of contents

Test level

To show how CrazyHook level looks like and how new logics work I prepared level which has name: !CHTest.WWD.

Once again some informations about parts of CH level:

a) **ANIS** - .ani files. I added CYCLE300.ani file and NEW folder, in this folder CYCLE150.ani file.

Example for CYCLE300.ani file - **CUSTOM_CYCLE300** (LEVEL_TORCHSTAND).

Example for file from NEW folder - **CUSTOM_NEW_CYCLE150** (LEVEL_HANDS4 in this level).

b) **IMAGES** - .**bmp**, .pid, .pcx files. I added SPLASH folder (it is empty, in La Roca Claw can die only because of spikes). In other levels we can use default images (they must be converted) or make new images and paste them to this folder. I added also NEW folder, in this folder FRAME1.bmp file.

Example for file from NEW folder - **CUSTOM_NEW** (ROBBERTHIEF from 3rd level).

c) **LEVEL** - like above, .wav files. Folders in this "folder" you can treat as curiosity because usually we use specific names. Difference between LEVEL folder and IMAGES folder is, that LEVEL folder we can use for default logics.

Example 1: stalactite from 12th level. I added PROJECTILES -> STALACTITE folders, in this folder stalactite images. In LEVEL folder I added STALACTITEHIT2.wav and STALACTITESNAP2.wav. We have to write this in **main.lua** logic:

```
function OnMapLoad()
local init = MapAnisFolder(LoadFolder("LEVEL12_ANIS_STALACTITE"),"LEVEL_STALACTITE")
end
```

Example 2: RobberThief from 3rd level. I added ROBBERTHIEF and ARROW folders. In first folder you can see images, .ani and .wav files. This is perfect example because all files are in one folder. We do not need to set anything with arrows for this opponent.

d) **LOGICS** - .lua files. In this folder you will find all new logics. Example - I added *Test* logic. It creates CrumblingPeg next to the start position. Other, new logics you will see on the next pages.

e) **MUSIC** - .xmi files. LEVEL.xmi file - new music in level. It will replace default soundtrack. I added also NEW.xmi file. Game will play this music if you will stand next to the skull (NewMusic.lua logic).

f) **SCREENS** - .pcx file. I added LOADING.PCX file.

g) **SOUNDS** - .wav files. I added SOUND1.wav file and NEW folder, in this folder SOUND2.wav file.

Example (1st file) - **CUSTOM_SOUND1** in Animation field (above LITTLEPUDDLE) .
Example (2nd file) - **CUSTOM_NEW_SOUND2** in Animation field (above FLOORCELL).

Using new sounds is very simple.

h) **TILES** - .bmp, .pid, .pcx files. I added to the ACTION folder two files: 015.bmp and 311.bmp (from 3rd level). 015.bmp will be displayed properly. Instead new 311.bmp it will be displayed default image from BASE level (La Roca). If new images have to be displayed properly, you have to name them using different names than default names.

I added to the NEW folder 5 new images for tiles (from 2nd level). I added also new layer (Z: 8500) so I did not have to change names to display this tiles properly.

a) Logic - structure

```
1   local variableA = 5
2
3   function init(self)
4       self.HitRect = {-32,-32,32,32}
5       self.HitTypeFlags = ObjectType.Special
6       --CODE
7   end
8
9   function main(self)
10      if self.State == 0 then
11          self.State = 1  --self.State = variableA
12          self.AttackRect = {-32,-32,32,32}
13          self.AttackTypeFlags = ObjectType.Player
14          --CODE
15      end
16  end
17
18  function attack(self)
19          --CODE
20  end
21
22  function hit(self)
23          --CODE
24  end
```

You can see above how structure of logic looks like. Of course it is only a scheme. Except main function, you do not need to use other functions if you do not want.

1) Before any function you can use own, local variables.

2) **init(self)** function - we use this function if we want to set something when logic is loading and before first state of the logic (0) will be changed to the next state.

3) **main(self)** function - in this function there are main variables and instructions. Logic without this function will not work.

4) **attack(self)** function - in this function we write what logic will do if object (player in this example) will enter to the attack area which was set in AttackRect.

5) **hit(self)** function - in this function we write what logic will do if object will be hit (one of the meaning). Hit area is set in HitRect.

6) We can write own functions and use them. In **main.lua** logic we can write global functions and global variables and then use them in new logics.

b) Claw inputs - basic values

Name	Hex	Dec
Nothing	0	0
Jump	1	1
Fist/leg hit	2	2
Last used weapon	4	4
Change weapon	8	8
Lift/throw	20	32
Pistol shot	40	64
Sabre attack	42	66
Magic claw	80	128
Dynamite	100	256
Special attack	200	512
Left	1.000.000	16.777.216
Right	2.000.000	33.554.432
Up	4.000.000	67.108.864
Down	8.000.000	134.217.728

There are more values which we can use. If you want to find other values e.g. 'Down' + 'Sabre attack' (Hex: 8.000.042, Dec: 134.217.794) you can:

1. Add both values: 134.217.728 + 66 = 134.217.794
2. Use GetInput() function.

Example - logic which we can use to get more values:

```
1  function main(self)
2      TextOut(GetInput())
3  end
```

ATTENTION!

GetInput() function allows to get only these values which are assigned to specific moves like left, punch or jump. You can not get values from all keys from keyboard.

c) VKeys - virtual keyboard

KeyPressed(arg) function allows to use all keys from keyboard. 'arg' - value in hexadecimal system. You can find these values on this site:
https://docs.microsoft.com/en-us/windows/win32/inputdev/virtual-key-codes

d) We are creating our first logic

As I wrote before, previous logics and logics which we will create will be in !CHTest level. Names for our logics: *Logic1*, *Logic2* etc.

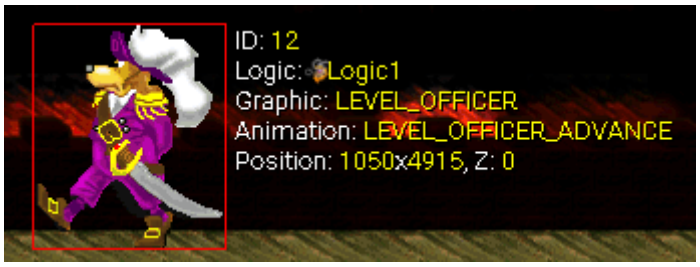
Two ways, how to add new logic in Editor:

a) **WapWorld** - create new object and type in Logic: field CustomLogic, then type e.g. *Logika1* name in Name: field.

b) **WapMap Beta** - just select Custom option: **Logic Type** and then type name in Logic field.

Logic name and **.lua** file name must be the same. We can edit LUA file with notepad (better option is using Notepad++).

Logic1 (Part I) - main function, movements and animation



We create object like you can see on this screenshot.

This is how logic looks like:

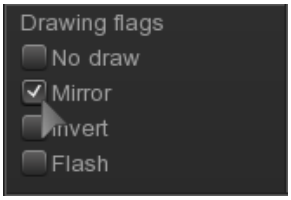
```
1  function main(self)
2      self:AnimationStep()
3  end
```

Now, check in game how the object will behave. So, what can you see? Object behaves like animated object. We just created own implementation of *AniCycle* logic ☺.

AnimationStep() function sets the right frame (I field in editor) for current animation. Because our logic (main function) is working all the time, we can see animation.

But what does it mean that our logic work all the time?

We will write in logic new line (between 2nd and 3rd): **self.X = self.X + 1**. How the object will behave? Object moves to the right. Officer looks like he is dancing MoonWalk ☺. We fix it by set Mirror flag in editor (Drawing flags).



You can see how to set Mirror flag in object properties.

Now, Officer is facing to the right way. Next line in our logic: **self.DrawFlags.Mirror = true**. Now we do not have to set Mirror flag every time in editor.

We can also set image and animation for this object in our logic. At first we will clean these fields in editor. We will use new functions: SetImage() and SetAnimation(). Our logic should look like this:

```
1  function main(self)
2      self:AnimationStep()
3      self.X = self.X + 1
4      self.DrawFlags.Mirror = true
5      self:SetImage("LEVEL_OFFICER")
6      self:SetAnimation("LEVEL_OFFICER_ADVANCE")
7  end
```

Check now how the object will behave.

Something is wrong ☹. Why the object has no longer animation?

In our logic we have logic problem. Before, I wrote that main function works all the time. It means logic tries play LEVEL_OFFICER_ADVANCE animation from first frame in every tick!

How we can solve this problem?

Logic1 (Part II) - object initialization, conditional statements

When we create logic for object, we want some things set only once - **initialization**. We will use function which you already know: **init(self)**. We will write this function above main function and we will move 3 elements: setting image, setting animation and Mirror flag. So, our logic should look like this:

```
function init(self)
    self:SetImage("LEVEL_OFFICER")
    self:SetAnimation("LEVEL_OFFICER_ADVANCE")
    self.DrawFlags.Mirror = true
end
function main(self)
    self:AnimationStep()
    self.X = self.X + 1
end
```

If you wrote everything right, congratulation! Officer has animation again ☺.

What if you would like to have logic which can set new image/animation but also possibility to set default image/animation? In this case we will use conditional statements.

a) At first we will write condition for default image: **if self.Image == nil then** (if we did not select image, then). This condition we write above function which sets image. At the end we have to close the function with **end** word.

b) Second condition for default animation: **if self.Animation == nil then** (if we did not select animation, then). This condition we write above function which sets animation. At the end we have to close the function with **end** word.

ATTENTION!

We test new logics using CrazyHook v1.4.4.4. It means that condition for default animation looks differently than in previous CrazyHook versions. In old versions condition for default animation looks like this: **if self._Animation_ == 0 then**.

If you wrote everything right, logic should look like this:

```
function init(self)
    if self.Image == nil then
        self:SetImage("LEVEL_OFFICER")
    end
    if self.Animation == nil then
        self:SetAnimation("LEVEL_OFFICER_ADVANCE")
    end
    self.DrawFlags.Mirror = true
end
function main(self)
    self:AnimationStep()
    self.X = self.X + 1
end
```

We will insert second Officer with the same X coordinate above first Officer. Second Officer will use the same logic like first Officer. We will make a small race ☺. It is time to modificate our code. Instead **self.X = self.X + 1** we write **self.X = self.X + self.SpeedX**. As you can see, we can use our values for speed.

In editor, for the first Officer we type in SpeedX field value 1, for second Officer, value 2. In the last step, we will set for second Officer LEVEL_OFFICER_FASTADVANCE animation (in editor). Now play the level and check which Officer will win (flag = finish) ☺.

For now, Officers are moving only in the right side. We will try to set borders using: XMin and XMax fields (in editor):

We can type values manually or by clicking Pick and selecting XMin and XMax.

We will set the same values for XMin and XMax. Play level again. When Officer reached border he went further. We have to write more instructions for logic. We have to tell the Officers what they to do when they reach border ☺.

Logic1 (Part III) - object states handling

Look at the table below. This is example how to write states for logic and what we can use after "==".

```
if self.State == 1 then
    self.X = self.X + 1
elseif self.State == 2 then
    self.X = self.X - 1
elseif self.State == 3 then
    self.Y = self.Y + 1
elseif self.State == 4 then
    self.Y = self.Y - 1
```

```
if self.State == GOING_RIGHT then
    self.X = self.X + 1
elseif self.State == GOING_LEFT then
    self.X = self.X - 1
elseif self.State == GOING_DOWN then
    self.Y = self.Y + 1
elseif self.State == GOING_UP then
    self.Y = self.Y - 1
```

Left side of the table - we use numbers. It is good option if logic is short and is not complex. But if we use local variables (right side of the table) code is more clearer. These labels are better because we know for what label (state) is responsible for.

So we will write above **init(self)**: **local GOING_RIGHT = 0** function, line below **local GOING_LEFT = 1**. **local** word before name of variable means that we can use this variable only in specified block in logic. It means also that this variable will be used only if we need it.

We will write now new instruction. It will set direction for Officer when logic has state = **0**. In **init(self)** function we write: **if self.Direction == 0 then** and next lines like at the screenshot below:

```
1  local GOING_RIGHT = 0
2  local GOING_LEFT = 1
3  function init(self)
4      if self.Image == nil then
5          self:SetImage("LEVEL_OFFICER")
6      end
7      if self.Animation == nil then
8          self:SetAnimation("LEVEL_OFFICER_ADVANCE")
9      end
10     if self.Direction == 0 then
11         self.State = GOING_RIGHT
12     else
13         self.State = GOING_LEFT
14     end
15     self.DrawFlags.Mirror = true
16 end
```

But what exactly are these ”states”?

Using simple words, states mean ”what this object is doing now?”. Our object will be doing two things. First state - moves to the right, second state - moves to the left. Logic can use many states, it depends from our invention. Once again, object after level loading has **state = 0**. **init()** function is like first logic state in which we write what should be set (similar to what we set in editor).

In main function we will write condition for moving to the right when logic **state = GOING_RIGHT** and second condition for moving to the left.

```
function main(self)
    self:AnimationStep()
    if self.State == GOING_RIGHT then
        self.X = self.X + self.SpeedX
    elseif self.State == GOING_LEFT then
        self.X = self.X - self.SpeedX
    end
end
```

We are so close to write full logic ☺.

Next task for us is write instructions, change direction if Officer will reach border:

a) Below **self.X = self.X + self.SpeedX** line we will write first condition: **if self.X >= self.XMax then** (if object reached value X greater than or equal to XMax then). Next line, changing object state: **self.State = GOING_LEFT** and last line, changing object Mirror flag to **false**.

b) Below **self.X = self.X - self.SpeedX** line we will write second condition: **if self.X <= self.XMin then** (if object reached value X smaller than or equal to XMin then). Next line, changing object state and Mirror flag to **true**. Full logic should look like this:

```
local GOING_RIGHT = 0
local GOING_LEFT = 1
function init(self)
    if self.Image == nil then
        self:SetImage("LEVEL_OFFICER")
    end
    if self.Animation == nil then
        self:SetAnimation("LEVEL_OFFICER_ADVANCE")
    end
    if self.Direction == 0 then
        self.State = GOING_RIGHT
    else
        self.State = GOING_LEFT
    end
    self.DrawFlags.Mirror = true
end
function main(self)
    self:AnimationStep()
    if self.State == GOING_RIGHT then
        self.X = self.X + self.SpeedX
            if self.X >= self.XMax then
                self.State = GOING_LEFT
                self.DrawFlags.Mirror = false
            end
    elseif self.State == GOING_LEFT then
        self.X = self.X - self.SpeedX
            if self.X <= self.XMin then
                self.State = GOING_RIGHT
                self.DrawFlags.Mirror = true
            end
    end
end
```

Check logic in game…

Congratulations!

You know how to create animated object, set movements for object and how to use object states. Setting Flags or default image/animation is easy for you.

Homework

Write logic for object which will not move only in horizontal way. If object state will change, image/animation will change. Add condition for default borders if you will forget set them in editor.

Challenge. Write logic similar to **Logic1** which will not use **init()** function and will not use **local** variables.

e) Officer which drops treasures and CreateObject function

Second logic will be using new functions. You will learn how to make more complex things. Our task is to create new object (officer) which will be animated and will do new activities. Goals:

a) officer after reaching border will do new animation

b) officer drops the coin, after that we change coin to other treasure

c) using CreateObject function logic will create CrumblingPeg to go further

Logic2 (Part I) - dropping treasure by object

We create new object, close to start position. In editor we set **SpeedX = 1**, **XMin** the same like X of new object and **XMax** above FLOORCELL object which I added to editor. I recommend these settings to focus only on new things.

In logic we set image and animation for walking. In this case officer will move only once, to the right border. Write condition, like before that, if officer will reach **XMax** then **state** will change and new animation will set - LEVEL_OFFICER_STRIKE. Example:

```
1  function main(self)
2      self:AnimationStep()
3      if self.State == 0 then
4          self.State, self.DrawFlags.Mirror = 1, true
5          self:SetImage("LEVEL_OFFICER")
6          self:SetAnimation("LEVEL_OFFICER_FASTADVANCE")
7      end
8      if self.State == 1 then
9          self.X = self.X + self.SpeedX
10             if self.X >= self.XMax then
11                 self.State = 2
12                 self:SetAnimation("LEVEL_OFFICER_STRIKE")
13             end
14     end
15 end
```

As you can see, if logic **state = 0** logic sets default elements of logic. I wrote two things in line 4 for shorter notation. You do not need to use this notation but it is good to know that something like this exists.

What actually happened? Officer reached border and did new animation. We will write another change for state and we will use new function: **DropCoin()** when logic state will be 3.

ATTENTION!

If you want to use variable, you should use **dot** after self, but if you want to use function, you should use **colon** after self.

We added coin drop function for officer. But what if we would add turnover for officer after animation and coin drop in specific moment?

We will check in editor which animation frame we should use. Frame 204 will be perfect. We will write instruction using two conditions if self.State == 2 **and self.I == 204**. It means that both conditions must be met. Last thing is change Mirror flag. If you have any problems, check screenshot below:

```
15      if self.State == 2 and self.I == 204 then
16          self.State, self.DrawFlags.Mirror = 3, false
17          self:DropCoin()
18      end
```

Why officer turnovered when animation was in a halfway?

We have to change it. Second condition should be like this ”is animation has finished?”. Instead **self.I == 204** we will use **self._unkn_bool2 ~= 0**. Now it works good. Do not focus on how second condition looks like for now ☺.

But what if we want other treasures, not only coins? What if we want the same treasure but in specific place like place where officer hits using his sabre? We will need **CreateGoodie(table)** function. Delete **DropCoin()** and use **CreateGoodie {x=self.X, y=self.Y, z=self.Z, powerup=33}**. As you can see we can choose treasure and set X, Y, Z coordinates. You do not need to write **self** in this line. So it is time to set place for coin drop. You can see full logic below:

```
1   function main(self)
2       self:AnimationStep()
3       if self.State == 0 then
4           self.State, self.DrawFlags.Mirror = 1, true
5           self:SetImage("LEVEL_OFFICER")
6           self:SetAnimation("LEVEL_OFFICER_FASTADVANCE")
7       end
8       if self.State == 1 then
9           self.X = self.X + self.SpeedX
10              if self.X >= self.XMax then
11                  self.State = 2
12                  self:SetAnimation("LEVEL_OFFICER_STRIKE")
13              end
14      end
15      if self.State == 2 and self._unkn_bool2 ~= 0 then
16          self.State, self.DrawFlags.Mirror = 3, false
17          CreateGoodie {x=self.X+100 , y=self.Y-25 , z=2000, powerup=33}
18      end
19  end
```

Logic3 (Part I) - CreateObject function

Your next task will be write logic which will allow bypass the spikes. We will add this logic close to the checkpoint flag. At first in main function we will set area in which Claw will active the logic and variable which will detect that the Claw (player) is in this area. It is not necessary but we will add image for this logic.

We change object state and image, I chose Catnip image. In the next line we write **self.AttackRect = {-32,-32,32,32}**. In this way we set area for player.

Values in curly brackets: how many pixels from object center to the left side (Left), how many pixels from object center to the top (Top), how many pixels from object center to the right side (Right), how many pixels from object center to the bottom (Bottom).

In the next line we write **self.AttackTypeFlags = ObjectType.Player**. In this way we set that only player can active logic. You may notice that the word **Player** has been written after dot. We can change to other objects e.g. Enemy. You can read about what you can write after dot in CrazyHook.lua (ObjectType) file which is in a main folder of Claw. Below **end** word, which ends main function we will write attack function (logic - structure, page 4th). This logic should look like this:

```
1  function main(self)
2      if self.State == 0 then
3          self.State = 1
4          self:SetImage("GAME_CATNIPS_NIP1")
5          self.AttackRect = {-32,-32,32,32}
6          self.AttackTypeFlags = ObjectType.Player
7      end
8  end
9  function attack(self)
10
11 end
```

Now we will write **CreateObject** function in **attack** function.

CreateObject {x=self.X, y=self.Y, z=self.Z, logic="CrumblingPeg", image="LEVEL_CRUMBLINGPEG"}.

x, y, z - coordinates for new object, logic - name of logic, image - image for new object. We change coordinates to new object has been created above spikes. Check logic.

Why logic created so many CrumblingPegs instead only one?

We can solve this problem using one of two ways:

1. Inside **attack** function, we change object state to 2.

```
9  function attack(self)
10     if self.State == 1 then
11         self.State = 2
12         CreateObject {x=1820, y=4950, z=1000, logic="CrumblingPeg", image="LEVEL_CRUMBLINGPEG"}
13     end
14 end
```

2. We use Destroy() function.

```
function attack(self)
    CreateObject {x=1820, y=4950, z=1000, logic="CrumblingPeg", image="LEVEL_CRUMBLINGPEG"}
    self:Destroy()
end
```

Now we have only one CrumblingPeg ☺.

We will try to use other logic to avoid death of falling on spikes. We will write *Elevator* logic, Elevator image and after image, SpeedX = 200, XMin = 1800, XMax = 1840. (XMin and XMax will depend on where we placed Elevator, CreateObject coordinates). Any problems? Check screenshot below:

```
CreateObject {x=1820, y=4950, z=1000, logic="Elevator", image="LEVEL_ELEVATORS", SpeedX=200, XMin=1800, XMax=1840}
```

Now you can bypass spikes using Elevator and collect the skull as your reward. As you can see writing logics is not that hard as you could think.

Congratulations!

You know how to create object which can drops treasures and you know more about animation. CreateObject function is easy to use for you and you know how to active logic in specific area.

Homework

Write logic (object) which will drop random treasures in random places when you will be (you will activate) in specific area.

Write logic (object) which will move to random place every time when you will start level again. Let logic changes state and animation when it will reach specific amount of pixels. To change object state to the next object state animation must be finished.

Challenge. Write logic A, if you will reach logic A area, let this logic create another logic B (set image for new logic B) and if you will be 100 pixels away, logic B will be destroyed.

f) Time, KeyPressed function

In the last part of this document I will show you how to use time and other keys from the keyboard than ”available” in GetInput function. You will use current knowledge and new things. Goals:

a) create new type of Elevator, changing state after time

b) change Elevator to other object

c) create object, using keys from keyboard to move new object

Logic4 (Part I) - new type of Elevator

We start from the main function. We set image and we use **self.HitRect** and **self.ObjectTypeFlags** to change object into the platform. You probably noticed that we do not use **self.HitTypeFlags**. We have to set type of object to *Special*. Try to write this logic, if you have any problems, check screenshot:

```
1  function main(self)
2      if self.State == 0 then
3          self.State = 1
4          self:SetImage("LEVEL_ELEVATORS")
5          self.HitRect = {-32,-8,36,8}
6          self.ObjectTypeFlags = ObjectType.Special
7      end
8  end
```

This logic we add next to the supercheckpoint (yellow flag, just jump into Warp). We can stay on new platform but for now platform can not move. Probably you know what to use to have working elevator. We will write condition, if Claw is on Elevator then Elevator will move. We will use **Object:IsBelow(object)** function. ”Object” before colon means elevator, second ”object” means Claw.

So, **self:IsBelow(GetClaw())**. In this case self means platform, GetClaw() function means Claw. You can see below how to write this condition:

```
8      if self.State == 1 and self:IsBelow(GetClaw()) then
9          self.X = self.X + 1
10     end
11 end
```

Check logic.

Platform can moves but Claw fell down, why?

We only added movement for elevator but not for Claw. It is simple to solve this problem, we will write **self.MoveClaw = 1** (or other value, which we add to self.X). Remember that both values have to be the same.

We will make some changes in second condition. We will write, if Claw will stay on platform, only state will change. Now it is time to add next object state and in this state platform will move. This is similar to TriggerElevator logic.

We will write two new variables after first condition, **self.XMax = self.X + 250** and **self.StopTime = 0**. First variable is a XMax border for platform, we will use second variable in a while. I recommend to use X: 3470 and Y: 5240 coordinates for this logic to focus on main goal. We will write another state change, if platform will reach border (XMax). If you have problems, look at this screenshot:

```
function main(self)
    if self.State == 0 then
        self.State = 1
        self.XMax = self.X + 250
        self.StopTime = 0
        self:SetImage("LEVEL_ELEVATORS")
        self.HitRect = {-32,-8,36,8}
        self.ObjectTypeFlags = ObjectType.Special
    end
    if self.State == 1 and self:IsBelow(GetClaw()) then
        self.State = 2
    end
    if self.State == 2 then
        self.MoveClawX = 1
        self.X = self.X + 1
            if self.X >= self.XMax then
                self.State = 3
            end
    end
end
```

Check logic.

In last state platform stopped but Claw no and he fell down on the spikes ☹, why?

We can solve this problem, just write below **self.State = 3** line **self.MoveClawX = 0**. Now Claw will not move when object state will change to last state.

We will write next condition in which platform will move to the top. You know now what to do. Below the first condition, like before we will write **self.YMin = self.Y - 500**. We will add next state, if platform will reach YMin border. Logic should look like this (next page):

```
function main(self)
    if self.State == 0 then
        self.State = 1
        self.XMax = self.X + 250
        self.YMin = self.Y - 500
        self.StopTime = 0
        self:SetImage("LEVEL_ELEVATORS")
        self.HitRect = {-32,-8,36,8}
        self.ObjectTypeFlags = ObjectType.Special
    end
    if self.State == 1 and self:IsBelow(GetClaw()) then
        self.State = 2
    end
    if self.State == 2 then
        self.MoveClawX = 1
        self.X = self.X + 1
            if self.X >= self.XMax then
                self.State = 3
                self.MoveClawX = 0
            end
    end
    if self.State == 3 then
        self.MoveClawY = -1
        self.Y = self.Y - 1
            if self.Y <= self.YMin then
                self.State = 4
                self.MoveClawY = 0
            end
    end
end
```

Now we will use **self.StopTime** variable. Below **self.MoveClawY = 0** line we will write **self.StopTime = GetTicks()**. We saved time and assigned to the new variable. You are right ☺, we will use this variable in the next condition, which will look like this: subtract time which is saved in **self.StopTime** variable from current time (we can use **GetTime()** function or **GetTicks()** function) and if time is greater than 5 seconds (*5000*) then change object state Try to write it but if you have problems, look at the screenshot:

```
    if self.State == 3 then
        self.MoveClawY = -1
        self.Y = self.Y - 1
            if self.Y <= self.YMin then
                self.State = 4
                self.MoveClawY = 0
                self.StopTime = GetTicks()
            end
    end
    if self.State == 4 and GetTime() - self.StopTime > 5000 then
        self.MoveClawX, self.MoveClawY = 3, 2
        self.X, self.Y = self.X + 3, self.Y + 2
            if self.Y >= 5150 then
                self.State = 1
                self.X, self.Y = 3470, 5240
                self.MoveClawX, self.MoveClawY = 0, 0
            end
    end
```

I added movement for elevator if state = 4 and if elevator will be higher than Y = 5150 then object will back to the start position (start position of logic). We sacrificed more time to write and understood logic but new knowledge will be useful for you in the future ☺.

Logic4 (Part II) - changing elevator into new object

Let's write again logic which will be an elevator. Leave **self.XMax** but change border to 750 pixels. We will write the same instruction, platform will move if Claw will be on platform. If you have any doubts, screenshot:

```
function main(self)
    if self.State == 0 then
        self.State = 1
        self.XMax = self.X + 750
        self:SetImage("LEVEL_ELEVATORS")
        self.HitRect = {-32,-8,36,8}
        self.ObjectTypeFlags = ObjectType.Special
    end
    if self.State == 1 and self:IsBelow(GetClaw()) then
        self.State = 2
    end
    if self.State == 2 then
        self.MoveClawX = 2
        self.X = self.X + 2
            if self.X >= self.XMax then
                self.State = 3
                self.MoveClawX = 0
            end
    end
end
```

We will add new condition, if Claw will be several dozen pixels away, platform will change into treasure. In the next line we will change elevator image to treasure image. I chose GAME_TREASURE_CHALICES_PURPLE. Now we will change position of treasure, about 175 pixels higher and we will change object type to 0.

We will do something similar to what you already know, if player will be in treasure area he will collect treasure. You know one way to do this: **self.AttackRect** and **attack(self)** function. This time you will learn another way. We will use logic coordinates and GetClaw() function. Check screenshot below:

```
    if self.State == 3 and (GetClaw().X > self.X + 200) then
        self.State = 4
        self.Y = self.Y - 175
        self.ObjectTypeFlags = 0
        self:SetImage("GAME_TREASURE_CHALICES_PURPLE")
    end
    if self.State == 4 and
    (GetClaw().X > self.X - 25) and (GetClaw().X < self.X + 25) and
    (GetClaw().Y < self.Y + 25) and (GetClaw().Y > self.Y - 25) then
        self.State = 5
    end
```

It is not a hard way but you must be careful when you use ”<”, ”>”. We set image for logic but Claw can not collect treasure, let's change it. We will write below self.State = 5, **self:Destroy()**. What will happen? We can ”collect” treasure but treasure does not have any sound and what is worse, we will not get any points. To play any sound, you can use **PlaySound()** function. Localization of sound is a function argument. PlaySound(”GAME_PICKUP2”) - I chose this sound. Points we will add using **GetClaw().Score** variable. We will add 2500 points (you can write any value, even negative value). If you have any doubts, check screenshot on the next page:

```
26      if self.State == 4 and
27      (GetClaw().X > self.X - 25) and (GetClaw().X < self.X + 25) and
28      (GetClaw().Y < self.Y + 25) and (GetClaw().Y > self.Y - 25) then
29          self.State = 5
30          PlaySound("GAME_PICKUP2")
31          GetClaw().Score = GetClaw().Score + 2500
32          self:Destroy()
33      end
```

This treasure is not the same like original but as exercise it should be enough. If you want you can add GLITTER and other things. Remember that **self:Destroy()** must be as last line in instruction/function.

Logic5 (Part I) - how to use keyboard keys to move object

We will create another logic and we will set image. We will use W, A, S, D keys and **KeyPressed()** function. Let's write simple condition ”if you press D key, then”. We have to use value in hexadecimal system as function argument. You can find informations about these values on some websites (look at 5th page of this document). This logic should look like this:

```
1   function main(self)
2       if self.State == 0 then
3           self.State = 1
4           self:SetImage("LEVEL_SKULL")
5       end
6       if self.State == 1 and KeyPressed(0x44) then
7           self.X = self.X + 1
8       end
9   end
```

We will add possibility to control object with other keys from keyboard. We will use also C key to change camera between Claw and this logic. We can use **CameraToPoint()** function. Coordinates X and Y are arguments of this function. Code:

```
6       if self.State == 1 and KeyPressed(0x44) then    --Right
7           self.X = self.X + 2
8       elseif self.State == 1 and KeyPressed(0x41) then--Left
9           self.X = self.X - 2
10      elseif self.State == 1 and KeyPressed(0x57) then--Up
11          self.Y = self.Y - 2
12      elseif self.State == 1 and KeyPressed(0x53) then--Down
13          self.Y = self.Y + 2
14      end
15      if self.State == 1 and KeyPressed(0x43) then    --Camera
16          CameraToPoint(self.X, self.Y)
17      end
```

Check logic.

We can move object but it is not comfortable. Camera does not follow the object ☹.

Let's add possibility to move object diagonally and we will write **CameraToPoint(self.X, self.Y)** function below every condition. Now logic should look like this (screenshot on the next page):

```
function main(self)
    if self.State == 0 then
        self.State = 1
        self:SetImage("LEVEL_SKULL")
    end
    if self.State == 1 and KeyPressed(0x44) then    --Right
        self.X = self.X + 2
        CameraToPoint(self.X, self.Y)
            if KeyPressed(0x57) then                --Up & Right
                self.Y = self.Y - 2
            elseif KeyPressed(0x53) then            --Down & Right
                self.Y = self.Y + 2
            end
    elseif self.State == 1 and KeyPressed(0x41) then--Left
        self.X = self.X - 2
        CameraToPoint(self.X, self.Y)
            if KeyPressed(0x57) then                --Up & Left
                self.Y = self.Y - 2
            elseif KeyPressed(0x53) then            --Down & Left
                self.Y = self.Y + 2
            end
    elseif self.State == 1 and KeyPressed(0x57) then--Up
        self.Y = self.Y - 2
        CameraToPoint(self.X, self.Y)
    elseif self.State == 1 and KeyPressed(0x53) then--Down
        self.Y = self.Y + 2
        CameraToPoint(self.X, self.Y)
    end
end
```

You can say ”this logic does not look very neatly”, that's right, we will change it in a while. Now we will create last logic, only for Camera. Let's add new object to editor and call it ”*Camera*”. We will set **AlwaysActive** flag to use logic always.

In newest logic we will write condition for C key, we will use similar function: **CameraToObject()**. Argument of **GetObject(ID)** function is an ID number which you can find in editor (Logic5 ID). Camera logic, screenshot:

```
    if self.State == 1 and KeyPressed(0x43) then
        self.State = 2
        CameraToObject(GetObject(18))
    end
    if self.State == 2 and not KeyPressed(0x43) then
        self.State = 3
    end
--
    if self.State == 3 and KeyPressed(0x43) then
        self.State = 4
        CameraToClaw()
    end
    if self.State == 4 and not KeyPressed(0x43) then
        self.State = 1
    end
```

I added also more conditions ”if C key is not pressed”. I used them to have logic which reacts for key press, not for holding key. I used also **CameraToClaw()** function which I do not need to describe.

So, let's go back to Logic5 and let's write everything ”prettier”. Logic will work the same but now all things look a more clearer, screenshot:

```
function main(self)
    if self.State == 0 then
        self.State = 1
        self:SetImage("LEVEL_SKULL")
    end
    if self.State == 1 then
        if KeyPressed(0x57) then            --Up
            CameraToPoint(self.X,self.Y)
            self.Y = self.Y - 2
                if KeyPressed(0x44) then    --Right
                    self.X = self.X + 2
                elseif KeyPressed(0x41) then--Left
                    self.X = self.X - 2
                end
        elseif KeyPressed(0x53) then        --Down
            CameraToPoint(self.X,self.Y)
            self.Y = self.Y + 2
                if KeyPressed(0x44) then    --Right
                    self.X = self.X + 2
                elseif KeyPressed(0x41) then--Left
                    self.X = self.X - 2
                end
        elseif KeyPressed(0x44) then        --Right
            CameraToPoint(self.X,self.Y)
            self.X = self.X + 2
        elseif KeyPressed(0x41) then        --Left
            CameraToPoint(self.X,self.Y)
            self.X = self.X - 2
        end
    end
end
```

Now it looks better. I also changed comments.

Congratulations!

You mastered basics (even more ☺) how to create logics for CrazyHook levels. You know how process of creating logic looks like and you can use your new knowledge now. Make some new (crazy) things in your level. Good luck!

Homework

Write logic, if you will type your name, Claw will get any (random) Powerup. You can use **ClawGivePowerup(powerupID,time)** function. More informations you can find in CrazyHook.lua file.

Challenge. Use acquired knowledge to make crazy logic ☺.