Witaj! W tej części dokumentacji dowiesz się jak tworzyć logiki do swojego poziomu.
wersja: finalna, autor: Pejti
Podziękowania dla: Zax37, Kubus_PL

Spis treści

Poziom testowy

W celu pokazania jak wygląda poziom pod wersję 1.4+ oraz jak działają nowe logiki, przygotowałem poziom o nazwie !CHTest.WWD.

Jeszcze raz omówię poszczególne foldery dla stworzonego poziomu:

a) **ANIS** - pliki .ani. Został dodany plik CYCLE300.ani oraz folder NEW, a w nim plik CYCLE150.ani.

Przykład użycia pliku CYCLE300.ani - **CUSTOM_CYCLE300** (LEVEL_TORCHSTAND).

Przykład użycia pliku z folderu NEW - **CUSTOM_NEW_CYCLE150** (LEVEL_HANDS4 w stworzonym poziomie).

b) **IMAGES** - pliki .**bmp**, .pid, .pcx. Został dodany folder SPLASH (jest pusty ponieważ w La Roca nie można zginąć od utonięcia). W innym przypadku możemy użyć domyślnych grafik z innego poziomu (należy je wtedy skonwertować) lub stworzyć własne i je wkleić do tego folderu. Poza wklejeniem grafik nie trzeba nic ustawiać. Został również dodany folder NEW, a w nim plik FRAME1.bmp.

Przykład użycia pliku z folderu NEW - **CUSTOM_NEW** (ROBBERTHIEF z poziomu 3-go).

c) **LEVEL** - jak wyżej, .wav. Foldery, które zostały dodane należy traktować bardziej jako ciekawostkę, ponieważ zwykle używa się konkretnych nazw dla folderów i plików. Różnica pomiędzy LEVEL, a IMAGES jest taka, że w **LEVEL**, foldery możemy przypisać do domyślnych logik, które występują w grze.

Przykład 1: użycie stalaktytu z poziomu 12-tego. Został dodany folder PROJECTILES -> STALACTITE, a w nim grafiki dla stalaktytu. W folderze LEVEL zostały dodane pliki STALACTITEHIT2.wav i STALACTITESNAP2.wav. By móc w pełni skorzystać z tego rozwiązania, należy w logice **main.lua** napisać:

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

Przykład 2: użycie RobberThiefa z poziomu 3-go. Został dodany folder ROBBERTHIEF oraz folder ARROW. W tym pierwszym znajdują się nie tylko grafiki, ale również pliki .ani i dźwięki. Jest to doskonały przykład, ponieważ wystarczy wszystkie pliki wrzucić do jednego folderu. Dodatkowo nie trzeba nic ustawiać ze strzałami dla tego przeciwnika.

d) **LOGICS** - pliki .lua. W tym folderze znajdują się wszystkie nowe logiki. Dla przykładu dodałem nową logikę o nazwie *Test*. Jej zadaniem jest stworzenie CrumblingPega obok pozycji startowej poziomu. Kolejne logiki zostaną przedstawione na następnych stronach.

e) **MUSIC** - pliki .xmi. Plik LEVEL.xmi odpowiada za nową muzykę. Zastąpi on domyślny soundtrack w poziomie. Został dodany również plik NEW.xmi, zostanie on odtworzony, jeżeli staniesz w poziomie obok czaszki (logika NewMusic.lua).

f) **SCREENS** - plik .pcx. Został dodany plik LOADING.PCX.

g) **SOUNDS** - pliki .wav. Został dodany plik SOUND1.wav, a także folder NEW, a w nim plik SOUND2.wav.
Przykład użycia pierwszego pliku - **CUSTOM_SOUND1** w polu Animation (nad LITTLEPUDDLE).
Przykład użycia drugiego pliku - **CUSTOM_NEW_SOUND2** w polu Animation (nad FLOORCELL).

Jak widać na powyższych przykładach, używanie nowych dźwięków jest proste.

h) **TILES** - pliki .bmp, .pid, .pcx. Do folderu ACTION zostały dodane dwa pliki: 015.bmp oraz 311.bmp (z poziomu 3-go). W poziomie grafika 015.bmp zostanie wyświetlona poprawnie. Zamiast nowego 311.bmp zostanie wyświetlona domyślna grafika z BASE poziomu (La Roca). Jeżeli nowe grafiki dla kafelków mają zostać wyświetlone poprawnie, należy nazwać je używając innych nazw niż te, które zarezerwowane są dla domyślnych kafelków.

Do folderu NEW zostało dodanych 5 nowych grafik dla kafelków (z poziomu 2-go). Została dodana nowa warstwa (Z: 8500) by móc je wyświetlić poprawnie bez zmieniania nazw dla grafik.

a) Struktura logiki

```lua
local variableA = 5

function init(self)
    self.HitRect = {-32,-32,32,32}
    self.HitTypeFlags = ObjectType.Special
    --CODE
end

function main(self)
    if self.State == 0 then
        self.State = 1  --self.State = variableA
        self.AttackRect = {-32,-32,32,32}
        self.AttackTypeFlags = ObjectType.Player
        --CODE
    end
end

function attack(self)
        --CODE
end

function hit(self)
        --CODE
end
```

Na powyższym screenie została przedstawiona domyślna struktura logiki. Należy tę strukturę traktować jako schemat. Poza główną funkcją (main), nie jest konieczne używanie pozostałych funkcji, jeżeli nie będziemy z nich korzystać.

1) Przed funkcjami, możemy użyć własnych, lokalnych zmiennych zapisując je u góry.

2) Funkcja **init(self)** - korzystamy z niej, jeżeli chcemy ustawić pewne zmienne lub instrukcje podczas wczytywania logiki i zanim ta logika zmieni swój początkowy stan (0) na inny.

3) Funkcja **main(self)** - główna funkcja bez której nowa logika nie zadziała. W niej znajdują się główne zmienne i instrukcje.

4) Funkcja **attack(self)** - w niej zapisujemy zmienne lub instrukcje, które zostaną wykonane, jeżeli jakiś obiekt (w tym przypadku gracz) wejdzie w pole działania tej funkcji określone w AttackRect.

5) Funkcja **hit(self)** - w niej zapisujemy zmienne lub instrukcje, które zostaną wykonane, jeżeli obiekt z tą logiką zostanie uderzony (nie tylko). Obszar działania tej funkcji ustalamy w HitRect.

6) Możemy tworzyć własne funkcje i odwoływać się do nich. Dodatkowo w logice **main.lua** możemy tworzyć zmienne i funkcje globalne, z których będziemy mogli korzystać w nowo napisanych logikach.

b) Claw inputs - podstawowe

Nazwa	Hex	Dec
Nic	0	0
Skok	1	1
Uderzenie pięścią/nogą	2	2
Ostatnio użyta broń	4	4
Zmiana broni	8	8
Podnoszenie/rzut	20	32
Strzał z pistoletu	40	64
Cios/atak szablą	42	66
Magiczny pazur	80	128
Dynamit	100	256
Atak specjalny	200	512
W lewo	1.000.000	16.777.216
W prawo	2.000.000	33.554.432
W górę	4.000.000	67.108.864
W dół	8.000.000	134.217.728

Nie są to wszystkie wartości jakich możemy użyć. Dlatego też, aby poznać inne, np. połączenie ‘W dół’ + ‘Cios/atak szablą’ (Hex: 8.000.042, Dec: 134.217.794) należy:

1. Dodać do siebie wartości: 134.217.728 + 66 = 134.217.794
2. Skorzystać z funkcji GetInput().

Przykładowa logika, która pozwoli nam poznać wartości wciskanych klawiszy:

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

UWAGA!

Funkcja GetInput() pozwala poznać wartości klawiszy, które przypisane są do ruchu czy innych czynności, które może wykonać Claw. Nie można za jej pomocą sprawdzić wartości klawiszy, które odpowiadają za nic.

c) VKeys - klawiatura wirtualna

Funkcja, która pozwala użyć dowolnego klawisza z klawiatury nazywa się KeyPressed(arg). Za ‘arg’ czyli argument podstawiamy wartość klawisza w postaci Hex (heksadecymalnej). Wartość tą możemy znaleźć korzystając ze strony:
https://docs.microsoft.com/en-us/windows/win32/inputdev/virtual-key-codes

d) Tworzymy naszą pierwszą logikę

Jak wcześniej wspomniałem, poprzednie logiki oraz te, które będziemy tworzyć będą znajdować się w poziomie !CHTest. Logiki będziemy nazywać kolejno *Logic1*, *Logic2* etc.

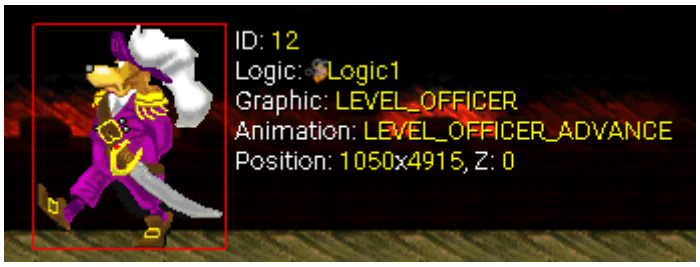
Sposoby dodania nowej logiki w zależności od wybranego edytora:

a) **WapWorld** - tworzymy nowy obiekt i wpisujemy w polu Logic: CustomLogic, a następnie nazwę np. *Logika1* w polu Name:.

b) **WapMap Beta** - wystarczy wybrać opcję Custom przy wyborze typu logiki **Logic Type**, a następnie wpisać jej nazwę w polu Logic.

Nazwa logiki oraz nazwa pliku **.lua** musi być taka sama. Plik ten możemy edytować notatnikiem (polecam korzystać z Notepad++).

Logic1 (Cz. I) - główna funkcja, podstawy ruchu i animacji



Tworzymy obiekt taki sam, jak ten widoczny na obrazku obok.

Tak prezentuje się logika:

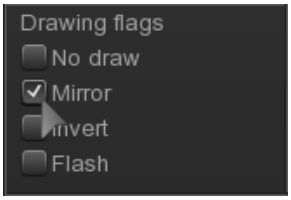
```
1  function main(self)
2      self:AnimationStep()
3  end
```

Sprawdźmy teraz w grze jak zachowa się ten obiekt. Co się właściwie dzieje? Stworzony obiekt zachowuje się jak obiekt animowany. Utworzyliśmy właśnie własną implementację logiki *AniCycle* ☺.

Funkcja **AnimationStep()** ustawia właściwą grafikę (pole I w edytorze) dla aktualnie odtwarzanej animacji. Ponieważ nasza logika (funkcja main) jest wykonywana cały czas, uzyskujemy płynność animacji.

Ale co to właściwie znaczy, że logika jest wykonywana cały czas?

Dopiszemy teraz do naszej logiki nowy wiersz: (między 2, a 3) **self.X = self.X + 1**. Jak zachowa się obiekt w grze? Obiekt zaczął przesuwać się w prawo. Wygląda to tak, jakby oficer tańczył MoonWalk ☺. Poprawmy to ustawiając flagę Mirror (odbicie lustrzane) w edytorze (flagi rysowania).



Ustawiamy flagę Mirror we właściwościach, w stworzonym przez nas obiekcie.

Teraz oficer zwrócony jest w odpowiednią stronę. Dopiszmy w naszej logice kolejny wiersz: **self.DrawFlags.Mirror = true**, aby obiekt z tą logiką od razu miał ustawioną flagę Mirror bez ustawiania jej za każdym razem w edytorze.

Możemy również ustawić za pomocą kodu grafikę i animację dla obiektu. Najpierw wyczyśćmy te pola w edytorze. Skorzystamy z dwóch nowych funkcji: SetImage() (ustawia grafikę) oraz SetAnimation() (ustawia animację). Tak powinna wyglądać nasza logika:

```
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
```

Sprawdźmy w grze jak zachowa się stworzony przez nas obiekt.

Coś jest nie tak ☹. Dlaczego obiekt już się nie animuje?

W kodzie pojawił się pewien problem logiczny. Wcześniej napisałem, że nasza funkcja działa przez cały czas. Oznacza to, że kod bez przerwy próbuje odtworzyć animację LEVEL_OFFICER_ADVANCE, rozpoczynając ją od początku!

Co z tym możemy zrobić?

Logic1 (Cz. II) - inicjalizacja obiektu, instrukcje warunkowe

Tworząc nową logikę dla obiektu, pewne rzeczy chcemy wykonać/ustawić tylko raz tzn. **zainicjalizować**. Skorzystamy z poznanej wcześniej funkcji **init(self)**. Dopiszemy teraz tę funkcję (nad główną funkcją) i przeniesiemy do niej 3 elementy: ustawianie grafiki, animacji i flagi rysowania Mirror. Po wykonaniu tych czynności nasza logika powinna wyglądać tak:

```
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
```

Jeżeli wszystko dobrze napisałeś, gratuluję! Oficer znów jest animowany ☺.

A co jeśli chcielibyśmy mieć możliwość ustawienia grafiki i animacji, ale także żeby logika ustawiała te pola domyślnie, jeżeli zapomnimy ustawić je w edytorze? W tym przypadku skorzystamy z instrukcji warunkowych.

a) Najpierw napiszemy warunek dla domyślnej grafiki: **if self.Image == nil then** (jeżeli nie wybraliśmy grafiki to). Ten warunek zapisujemy nad funkcją ustawiającą grafikę. Na koniec musimy domknąć ten warunek pisząc **end** pod tą samą funkcją.

b) Drugi warunek dla domyślnej animacji będzie wyglądać następująco: **if self.Animation == nil then** (jeżeli nie wybraliśmy animacji to). Ten warunek zapisujemy nad funkcją ustawiającą animację. Ten warunek również musimy domknąć słowem **end**.

UWAGA!

Poziom oraz logiki testowane są na wersji CrazyHook v1.4.4.4. Oznacza to, że warunek dla domyślnej animacji wygląda inaczej. W starszych wersjach gry warunek ten wygląda następująco: **if self._Animation_ == 0 then**.

Jeżeli wszystko napisałeś dobrze, logika powinna wyglądać tak:

```
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
```

Wstawimy teraz drugiego oficera o tej samej współrzędnej X znajdującym się nad pierwszym oficerem. Będzie on korzystał z tej samej logiki co pierwszy oficer. Zrobimy sobie taki mały wyścig oficerów ☺. Zmodyfikujemy troszkę nasz kod. Zamiast **self.X = self.X + 1** zmienimy na **self.X = self.X + self.SpeedX**. Jak już się domyślasz, sami będziemy mogli ustalać wartości dla poruszania się obu oficerów.

Wpisujemy w edytorze dla pierwszego oficera w polu SpeedX wartość 1, a dla drugiego oficera wartość wynoszącą 2. Ostatnim krokiem będzie ustawienie dla drugiego oficera animacji LEVEL_OFFICER_FASTADVANCE (w edytorze). Po ustawieniu wszystkiego włącz poziom i sprawdź, który oficer pierwszy dotrze do flagi i wygra wyścig ☺.

Jak na razie nasi oficerowie szli tylko w prawo. Spróbujmy ograniczyć obszar przesuwania się obiektów ustawiając pola: XMin i XMax (w edytorze):

Możemy wartości wpisać ręcznie lub kliknąć w przycisk Pick i wybrać XMin i XMax za pomocą myszki.

Ustawmy dla obu oficerów te same wartości XMin i XMax. Zobaczmy co się stanie. Oficer, gdy doszedł do XMax nie zareagował - poszedł sobie dalej. Dlatego też musimy uzupełnić logikę o nowe instrukcje. Powiedzieć mu co ma zrobić, gdy dojdzie do granicy ☺.

Logic1 (Cz. III) - obsługa stanów obiektu

Zanim przejdziemy do naszej logiki, popatrz na poniższą tabelę. Jest to przykład jak zapisywać stany dla logiki, a także co możemy wstawić po drugiej stronie równania.

```
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
```

Po lewej stronie tabeli zostały użyte liczby. Jest to dobra opcja, jeżeli nasza logika jest krótka i nie jest złożona. Jednak korzystając ze zmiennych lokalnych (po prawej stronie tabeli) będziemy mogli napisać kod, który jest bardziej czytelny. Takie etykiety dadzą nam informacje za co odpowiada konkretny stan logiki.

Zapiszmy zatem nad funkcją **init(self)**: **local GOING_RIGHT = 0**, a wiersz niżej **local GOING_LEFT = 1**. Słowo **local** przed nazwą zmiennej oznacza, że będziemy mogli użyć tej zmiennej tylko w określonym bloku naszej logiki. Oznacza również, że zmienna używana jest tylko wtedy kiedy jej potrzebujemy.

Napiszemy teraz warunek, od którego będzie zależeć, w którą stronę skierowany będzie oficer, kiedy nasza logika zostanie wczytana. W funkcji **init(self)** dopiszmy: **if self.Direction == 0 then**, a następnie kolejne wiersze tak jak na poniższym screenie:

```
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
```

A czym właściwie są te ”stany”?

Intuicyjnie możemy stwierdzić, że stan logiki odpowiada na pytanie ”co w tej chwili robi obiekt?”. Obiekt, dla którego piszemy logikę będzie wykonywać dwie rzeczy: przesuwać się w prawo oraz w lewo. Od nas zależy z ilu stanów będzie korzystać logika. Jeszcze raz zaznaczę, że obiekt z logiką po wczytaniu ma **stan = 0**. Funkcja **init()** to nic innego jak stan zerowy, w którym zapisujemy to co dla domyślnych logik ustawiamy w edytorze.

Przejdźmy do głównej funkcji naszej logiki i zapiszmy warunek, żeby oficer szedł w prawo, gdy stan logiki = **GOING_RIGHT**, a także warunek dla chodzenia w lewo.

```
17  function main(self)
18      self:AnimationStep()
19      if self.State == GOING_RIGHT then
20          self.X = self.X + self.SpeedX
21      elseif self.State == GOING_LEFT then
22          self.X = self.X - self.SpeedX
23      end
24  end
```

Jesteśmy coraz bliżej napisania pełnej logiki ☺.

Teraz czeka na nas zadanie podobne do poprzedniego. Musimy zapisać warunki, które powiedzą oficerowi, że musi zmienić kierunek marszu:

a) Pod wierszem **self.X = self.X + self.SpeedX** napiszemy pierwszy warunek: **if self.X >= self.XMax then** (jeśli obiekt osiągnął wartość X większą lub równą XMax to). Pod warunkiem zmieniamy stan logiki: **self.State = GOING_LEFT**, a linijkę niżej zmieniamy flagę Mirror na **false**.

b) Pod wierszem **self.X = self.X - self.SpeedX** napiszemy drugi warunek: **if self.X <= self.XMin then** (jeśli obiekt osiągnął wartość X mniejszą lub równą XMin to). Teraz pozostaje nam zmienić stan logiki oraz flagę Mirror na przeciwną. Cała logika powinna wyglądać następująco:

```
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
17  function main(self)
18      self:AnimationStep()
19      if self.State == GOING_RIGHT then
20          self.X = self.X + self.SpeedX
21              if self.X >= self.XMax then
22                  self.State = GOING_LEFT
23                  self.DrawFlags.Mirror = false
24              end
25      elseif self.State == GOING_LEFT then
26          self.X = self.X - self.SpeedX
27              if self.X <= self.XMin then
28                  self.State = GOING_RIGHT
29                  self.DrawFlags.Mirror = true
30              end
31      end
32  end
```

Sprawdźmy czy wszystko działa…

Gratulacje!

Potrafisz stworzyć obiekt, który jest animowany, porusza się, a także wykorzystuje stany. Poznałeś sposób jak ustawiać własną lub domyślną grafikę/animację oraz jak zmieniać flagi.

Zadanie domowe

✓ Napisz logikę, która sprawi, że obiekt będzie poruszać się nie tylko w poziomie, a kiedy zmieni się stan logiki, zostanie również zmieniona grafika. Dodaj warunki na wypadek, gdybyś zapomniał ustawić granice dla poruszającego się obiektu.

✓✓ Zadanie dla ambitnych. Napisz logikę podobną do **Logic1**, która nie będzie korzystać z funkcji **init()**, a także nie będzie korzystać ze zmiennych **lokalnych**.

e) Oficer upuszczający skarby oraz funkcja CreateObject

Druga logika będzie wykorzystywać nowe funkcje. Nauczysz się tworzyć bardziej złożone rzeczy. Naszym zadaniem będzie stworzyć obiekt (oficer), który będzie animowany i wykona nowe czynności. Cele do zrealizowania:

a) oficer dochodzi do granicy i wykonuje nową animację

b) zostaje upuszczony pieniążek, a następnie zmieniamy go na inny dowolny skarb

c) za pomocą funkcji CreateObject tworzymy CrumblingPega, by przejść dalej

Logic2 (Cz. I) - upuszczanie skarbu przez obiekt

Tworzymy nowy obiekt, najlepiej blisko miejsca, gdzie zaczynamy poziom. W edytorze ustawiamy wartość pola **SpeedX = 1**, wartość pola **XMin** taka sama jak współrzędna X obiektu, a wartość pola **XMax** kończąca się nad obiektem FLOORCELL, który jest już wstawiony do poziomu. Są to ustawienia preferowane, aby skupić się tylko na nowych rzeczach.

W naszej logice ustawiamy grafikę oraz animację dla chodzenia. W tym przypadku oficer przejdzie się tylko raz w prawą stronę. Napisz warunek, tak jak poprzednio, że jak oficer dojdzie do **XMax** to nastąpi zmiana stanu na **kolejny**, a także zostanie zmieniona animacja na LEVEL_OFFICER_STRIKE. Przykład logiki:

```
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
```

Jak widzisz powyżej, przy **stanie logiki = 0** następuje ustawienie podstawowych elementów. Został wprowadzony krótszy zapis w wierszu nr 4. Nie musisz go stosować, ale warto zapamiętać, że istnieje taka możliwość.

Co właściwie się stało? Oficer doszedł do granicy i wykonał nową animację. Dopiszmy kolejną zmianę stanu i nową funkcję **DropCoin()** przy zmianie stanu logiki na wartość 3.

UWAGA!

Gdy chcemy użyć zmiennej stawiamy **kropkę** po słowie self, natomiast gdy chcemy użyć funkcji to stawiamy **dwukropek** po słowie self.

Dodaliśmy upuszczenie pieniążka. Co jeśli chcielibyśmy, aby oficer obrócił się po wykonaniu animacji, a pieniążek został upuszczony w konkretnym momencie?

Najpierw sprawdźmy w edytorze, której klatki animacji powinniśmy użyć. Ta z numerem 204 będzie idealna. Musimy dopisać do warunku przy zmianie stanu z 2 na 3 słowo **and**, a po nim drugi warunek: **self.I == 204** (czy klatka animacji = 204). Oznacza to, że oba warunki muszą być spełnione jednocześnie. Dopiszmy też, że musi zostać zmieniona flaga Mirror. Tak powinien wyglądać powyższy warunek:

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

Dlaczego oficer obrócił się w połowie wykonywanej animacji?

Wygląda na to, że musimy zmienić warunek. Powinien on brzmieć ”czy animacja wykonała się do końca?”. Zamiast **self.I == 204** użyjemy **self._unkn_bool2 ~= 0**. Tym razem moneta wypadła dopiero po wykonaniu całej animacji. Na razie nie przejmuj się tym jak ten warunek wygląda ☺.

Co jeśli chcemy, aby został upuszczony inny skarb? A może ten sam, ale w miejscu, gdzie oficer ”uderza” szablą? Do tego będziemy potrzebować funkcji **CreateGoodie(table)**. Usuń funkcję **DropCoin()** i zastąp ją tym wierszem: **CreateGoodie {x=self.X, y=self.Y, z=self.Z, powerup=33}**. Jak możesz zaobserwować, ta funkcja pozwala nie tylko wybrać skarb, ale także ustawić współrzędne X, Y, Z. Dodatkowo nie musisz pisać słowa **self** w tym wierszu. Spróbujmy ustawić wypadanie pieniążka. Poniżej możesz zobaczyć logikę:

```
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 (Cz. I) - funkcja CreateObject

Twoim następnym zadaniem będzie napisanie logiki, która pozwoli nam ominąć kolce, które zapewne zauważyłeś już w poziomie. Dodamy ją w pobliżu flagi oznaczającej Checkpoint. Na początku, w głównej funkcji ustawimy obszar, w który będziemy musieli wejść, a także zmienną, która wykryje, że to właśnie gracz wszedł w ten obszar. Nie jest to konieczne, ale dla tego przykładu ustawimy obrazek dla Logic3.

Standardowo zmieniamy stan logiki na 1, jako obrazek wybrałem zwykłego Catnipa. W kolejnym wierszu napiszmy **self.AttackRect = {-32,-32,32,32}**. W ten właśnie sposób określiliśmy obszar, w którym musimy się znaleźć.

Wartości, które wpisujemy w nawiasie klamrowym: ile pikseli w lewo od środka obiektu (Left), ile pikseli do góry od środka obiektu (Top), ile pikseli w prawo od środka obiektu (Right), ile pikseli w dół od środka obiektu (Bottom).

W wierszu poniżej dopiszmy **self.AttackTypeFlags = ObjectType.Player**. W ten sposób ustawiliśmy, że gracz będzie mógł aktywować logikę. Zwróć uwagę, że po prawej stronie, po kropce mamy zapisane **Player**. Możemy zmienić na inny obiekt, np. na Enemy i to właśnie przeciwnik będzie mógł aktywować logikę. To co możemy wpisać po kropce znajduję się na samym początku pliku CrazyHook.lua (ObjectType) znajdującym się w głównym katalogu gry. Pod słowem **end** kończącym główną funkcję logiki, napiszmy funkcję attack (struktura logiki - strona 4-ta). Tak powinna wyglądać napisana przez nas logika:

```
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
```

Teraz napiszmy funkcję **CreateObject** wewnątrz funkcji **attack**.

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

x, y, z to współrzędne dla nowego obiektu, logic to nazwa logiki, a image to obrazek dla nowego obiektu. Zmieńmy współrzędne, żeby obiekt pojawił się nad kolcami. Sprawdźmy jak zadziała nasza logika.

Dlaczego pojawiło tak dużo CrumblingPegów, a nie tylko jeden?

Możemy ten problem rozwiązać na dwa sposoby:

1. Wewnątrz funkcji **attack** zmieniamy stan logiki na 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. Wykorzystujemy metodę Destroy(), by zniszczyć naszą logikę.

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

Tym razem pojawił się tylko jeden CrumblingPeg ☺.

Spróbujemy teraz użyć innej logiki, aby ominąć kolce i nie dać się zabić. Wpiszmy logikę *Elevator*, zmieńmy obrazek, a po nim zapiszmy SpeedX = 200, XMin = 1800, XMax = 1840. (XMin i XMax zależeć będzie od tego, jakie wpisałeś współrzędne w CreateObject). W razie wątpliwości screen z poprawnym kodem znajduję się poniżej:

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

Tym razem pojawiła się platforma, z której możesz skorzystać i zdobyć czaszkę jako nagrodę. Jak widzisz, pisanie logik nie jest takie trudne i nie potrzebujesz przeznaczać zbyt wiele czasu.

Gratulacje!

Potrafisz stworzyć obiekt upuszczający różne skarby, a także dowiedziałeś się więcej na temat animacji. Wiesz już jak używać metody CreateObject oraz jak aktywować logikę w określonym obszarze.

Zadanie domowe

✓ Napisz logikę (obiekt), która będzie upuszczać losowe skarby w losowych miejscach w poziomie jeżeli wejdziesz w obszar aktywowania.

✓ Napisz logikę (obiekt), która będzie przy każdym uruchomieniu poziomu poruszać się w losowym kierunku. Niech co określoną ilość pikseli zmienia swój stan i animację. Aby stan zmienił się na kolejny, animacja musi wykonać się do końca.

✓✓ Zadanie dla ambitnych. Napisz logikę, po wejściu w obszar aktywowania zostanie stworzona nowa Customową logika (ustaw jej dowolną grafikę), a ta niech zostanie zniszczona, jeżeli gracz oddali się od niej na odległość większą niż 100 pikseli.

f) Pobieranie czasu oraz używanie metody KeyPressed

W ostatniej części tego dokumentu nauczę cię posługiwać się czasem oraz używania innych klawiszy niż te, które są ”dostępne” w funkcji GetInput. Wykorzystamy zdobytą już wiedzę, połączymy to co już wiesz z czymś nowym. Do zrobienia:

a) stworzyć nowy rodzaj Elevatora (platformy), zmiana stanu po określonym czasie

b) zamienić napisany Elevator na inny obiekt

c) stworzyć obiekt, którym będziesz mógł sterować

Logic4 (Cz. I) - nowy typ platformy

Zaczynamy od głównej funkcji. Ustawmy grafikę dla naszej platformy oraz wykorzystajmy **self.HitRect**, a także **self.ObjectTypeFlags**, aby zamienić obiekt w coś na czym postawimy Clawa. Od razu pewnie zauważyłeś, że nie korzystamy z **self.HitTypeFlags**. Musimy ustawić typ obiektu na *Special*, aby wszystko zadziałało prawidłowo. Spróbuj teraz napisać tę logikę, a w razie problemu zobacz poniższy screen:

```
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
```

Naszą logikę wstawiamy obok złotej flagi (wskocz w Warp). Na platformie możemy już stać, ale na razie stoi ona w miejscu. Prawdopodobnie już wiesz czego użyć, by wprawić w ruch nasz Elevator. Napiszmy warunek, że ruch nastąpi dopiero, kiedy na nią wskoczymy. Wykorzystamy metodę **Object:IsBelow(object)**. Pierwsze słowo ”Object” oznacza, na którym obiekcie będziemy stać, a drugie słowo ”object”, oznacza obiekt stojący na tym pierwszym.

Innymi słowy napiszmy **self:IsBelow(GetClaw())**. W tym wypadku self to nasza platforma, a funkcja GetClaw() oznacza Clawa. Poniżej możesz zobaczyć poprawnie napisany warunek:

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

Sprawdźmy jak zadziała logika.

Platforma zaczęła poruszać się, ale Claw z niej spadł, dlaczego?

Problem polega na tym, że nie ustawiliśmy w logice, aby wraz z platformą poruszał się Claw. Wystarczy dopisać pod warunkiem **self.MoveClaw = 1** (lub inna wartość, która jest

dodawana do self.X). Pamiętaj, aby wartości były takie same, inaczej będziesz musiał kontrolować ruch Clawa.

Przerobimy teraz trochę drugi warunek. Napiszmy, że jak Claw stanie na platformie, to zmieni się jedynie stan logiki. Dopiszmy kolejny warunek i tam napiszmy, aby platforma poruszała się. W taki sposób stworzymy logikę TriggerElevator.

Dopiszemy po pierwszym warunku dwie zmienne, **self.XMax = self.X + 250** oraz **self.StopTime = 0**. Pierwsza to po prostu oznaczenie granicy XMax dla platformy, a drugiej użyjemy za chwilę. Proponuję, aby logika znajdowała się we współrzędnych X: 3470 i Y: 5240. Pod ostatnim warunkiem, dopiszmy zmianę stanu logiki, kiedy platforma osiągnie XMax. Jeżeli masz jakieś wątpliwości, skorzystaj z poniższego screena:

```
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
```

Sprawdźmy czy wszystko działa jaka należy.

Platforma przestała się poruszać, gdy stan zmienił się na ostatni, ale Claw nie i spadł na kolce ☹, dlaczego?

Możemy temu zaradzić dopisując pod **self.State = 3** wiersz **self.MoveClawX = 0**. Teraz Claw nie będzie się poruszać w osi poziomej.

Dopiszmy kolejny warunek, w którym platforma zacznie poruszać się do góry. Wiesz już co napisać, by Claw poruszał się tak jak platforma. Następnie w pierwszym warunku zapiszemy zmienną **self.YMin = self.Y - 500**. Dodajmy też warunek, aby platforma zatrzymała się, gdy osiągnie YMin. Logika obecnie powinna wyglądać tak (następna strona):

```
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
```

Teraz użyjemy zmiennej **self.StopTime**. Pod wierszem **self.MoveClawY = 0** dopiszemy wiersz zawierający **self.StopTime = GetTicks()**. W ten sposób zapisaliśmy aktualny czas w zmiennej po zmianie stanu logiki na stan 4. Tak, dobrze myślisz ☺, użyjemy tej zmiennej w kolejnym warunku, który będzie wyglądać następująco: odejmij czas zapisany w zmiennej **self.StopTime** od aktualnego czasu (możemy użyć zarówno funkcji **GetTime()** jak i **GetTicks()**) i jeżeli ten czas jest większy od 5-ciu sekund (*5000*) to zmień stan logiki na kolejny stan. Spróbuj napisać ten warunek, ale jeżeli nie wiesz jak, screen:

```
    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
```

Tak powinien wyglądać ten warunek. Dopisałem ruch dla platformy, gdy logika ma stan 4 oraz, aby po przekroczeniu Y = 5150 powracała na miejsce startu. Tym razem musieliśmy poświęcić więcej czasu na napisanie i zrozumienie logiki, lecz ta nowa wiedza na pewno ci się przyda w przyszłości ☺.

Logic4 (Cz. II) - zamiana platformy na inny obiekt

Napiszmy jeszcze raz logikę, która będzie platformą. Zostawmy **self.XMax**, zwiększając granicę o 500 pikseli. Niech platforma zacznie ruszać dopiero, kiedy na nią wskoczymy. W razie wątpliwości, zobacz poniższy screen:

```
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
```

Dodamy teraz warunek, taki że Claw musi odejść kawałek od platformy, aby ta zamieniła się w skarb. Pod nim napiszmy, aby grafika Elevatora zmieniła się na grafikę skarbu. Ja wybrałem GAME_TREASURE_CHALICES_PURPLE (różowy puchar). Dodatkowo zmienimy wartość Y dla skarbu o 175 pikseli w górę, a także zmienimy typ obiektu na 0.

Napiszmy zmianę stanu oraz warunek, że jeżeli gracz będzie w obszarze skarbu to go zbierze. Wcześniej używaliśmy **self.AttackRect** i funkcji **attack(self)**. Tym razem pokażę ci inną możliwość. Posłużymy się współrzędnymi logiki, a także funkcją GetClaw(), którą już znasz. Warunek będzie wyglądać następująco:

```
    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
```

Nie ma tu nic trudnego, jedynie trzeba uważać na znaki ”<”, ”>” ponieważ można się łatwo pomylić.

Okej, mamy grafikę, ale Claw na razie nie może zebrać skarbu, zmieńmy to. Dopiszmy pod stanem 5 znane ci **self:Destroy()**. Co się teraz stanie? Możemy ”zebrać” skarb, ale nie ma dźwięku, a co najważniejsze, nie dostaniemy za niego żadnych punktów. Odtwarzanie dźwięku odbywa się za pomocą funkcji **PlaySound()**. Jako argument musimy podać lokalizację dźwięku. PlaySound(”GAME_PICKUP2”) - taki wybrałem dźwięk dla skarbu. Punkty dodamy sobie przy pomocy zmiennej **GetClaw().Score**. Za puchar dostajemy 2500 punktów, więc i takiej wartości użyjemy (możesz wpisać dowolną wartość, nawet ujemną). Jeżeli masz wątpliwości to zobacz screen na następnej stronie:

```
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
```

Nie jest to taki sam skarb jak te, które można wstawić domyślnie do poziomu, ale jako ćwiczenie taka logika powinna wystarczyć. Jeżeli chcesz, możesz spróbować dodać poświatę dla skarbu (GLITTER), a także mały obrazek punktów, który zawsze pojawia się kiedy zbierzemy skarb (pojawia się, przesuwa do góry i znika). Zapamiętaj też, aby metoda **self:Destroy()** znajdowała się na samym końcu - ostatnia rzecz do wykonania.

Logic5 (Cz. I) - poruszanie obiektem za pomocą klawiszy

Stwórzmy nową logikę i ustawmy jej jakąś grafikę. Użyjemy klawiszy W, A, S, D do sterowania oraz funkcji **KeyPressed()**. Napiszmy prosty warunek, aby poruszyć obiektem w prawo - ”jeżeli wciśnięty jest klawisz D to”. Argumentem funkcji sprawdzającej, który klawisz został wciśnięty jest wartość podana w postaci heksadecymalnej. Wartość dowolnego klawisza możemy znaleźć w internecie (zobacz stronę 5-tą tego dokumentu). Tak będzie wyglądać logika:

```
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
```

Dodajmy możliwość sterowania w pozostałe kierunki. Wykorzystamy również klawisz C, aby zmieniać kamerę pomiędzy naszą logiką, a Clawem. Użyjemy do tego funkcji **CameraToPoint()**. Argumentami tej funkcji są współrzędne X i Y. Poniżej kod:

```
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
```

Sprawdźmy jak zadziała nasza logika.

Można poruszać obiektem, ale nie jest to komfortowe. Kamera nie podąża za obiektem ☹.

Najpierw dodajmy możliwość sterowania obiektem na skos w każdym kierunku, a następnie funkcję **CameraToPoint(self.X, self.Y)** napiszmy pod każdym warunkiem. Teraz logika wygląda następująco:

```
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
```

Zgodzisz się ze mną, że napisana logika nie wygląda zbyt schludnie. Wrócimy do tego za chwilę. Teraz użyjemy oddzielnej logiki, która będzie kontrolować przejście kamery pomiędzy Clawem, a naszą logiką. Dodajmy do edytora nowy obiekt i nazwijmy go ”*Camera*”. Ustawimy mu flagę **AlwaysActive**, aby zawsze móc z niego korzystać.

W nowo powstałej logice napiszmy warunek na wciśnięty klawisz C, a pod nim będzie podobna funkcja, **CameraToObject()**. Argumentem będzie funkcja **GetObject(ID)**, gdzie nr ID możemy sprawdzić w edytorze (Logic5 ID). Logika Camera, screen:

```
    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
```

Zostały dopisane dodatkowe warunki na zmianę stanu logiki - ”jeżeli klawisz C nie jest wciśnięty”. Jest to spowodowane tym, aby logika reagowała na wciśnięcie klawisza, a nie jego przytrzymanie jak w przypadku poruszania się obiektu za pomocą klawiszy. Użyliśmy też funkcji **CameraToClaw()**, której nie trzeba wyjaśniać.

Wróćmy do logiki Logic5 i napiszmy ją nieco ”ładniej”. Jej działanie będzie takie samo, lecz sam wygląd będzie bardziej przejrzysty, screen:

```
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
```

Teraz wygląda to lepiej. Zmodyfikowane zostały również komentarze do warunków.

Gratulacje!

Opanowałeś podstawy (a nawet więcej ☺) tworzenia logik do wersji CrazyHook. Wiesz już jak wygląda proces pisania, a zdobytą wiedzę możesz śmiało wykorzystać przy tworzeniu swojego poziomu. Nie pozostaje ci nic innego jak tworzyć kolejne mniej lub bardziej złożone (szalone) rzeczy. Powodzenia!

Zadanie domowe

✓ Napisz logikę, gdzie po wpisaniu swojego imienia zostanie aktywowany dowolny (losowy) Powerup. Wykorzystaj do tego funkcję **ClawGivePowerup(powerupID,time)**. Poszukaj informacji o tej funkcji w pliku CrazyHook.lua w głównym katalogu gry.

✓✓ Zadanie dla ambitnych. Wykorzystaj zdobytą wiedzę, by napisać coś szalonego ☺.