

Introduction

EngineA (EA) is a sector (room) based 3D game engine. A level/map is a bunch of sectors (rooms) connected by portals (openings, doorways). By calculating the visibility of the portals in the sector that the player (camera) is located it's quite easy to determine which sectors need to be rendered. If the doorway to a room is not visible, that room needn't be rendered. And if the doorway is visible, the rendering can be clipped to the bounding rectangle of the doorway. In short, sectors and portals are used to speed up rendering, which is quite crucial for an engine written entirely in an interpreted and pretty slow programming language.

The files

The files for this tutorial are located in the N7/examples/enginea_library folder, and the map editor, enginea_editor.exe, is located in the N7 folder.

The document enginea.pdf lists all the functions in the enginea library, but I strongly, strongly, **STRONGLY** suggest you to look at this tutorial and the examples first of all.

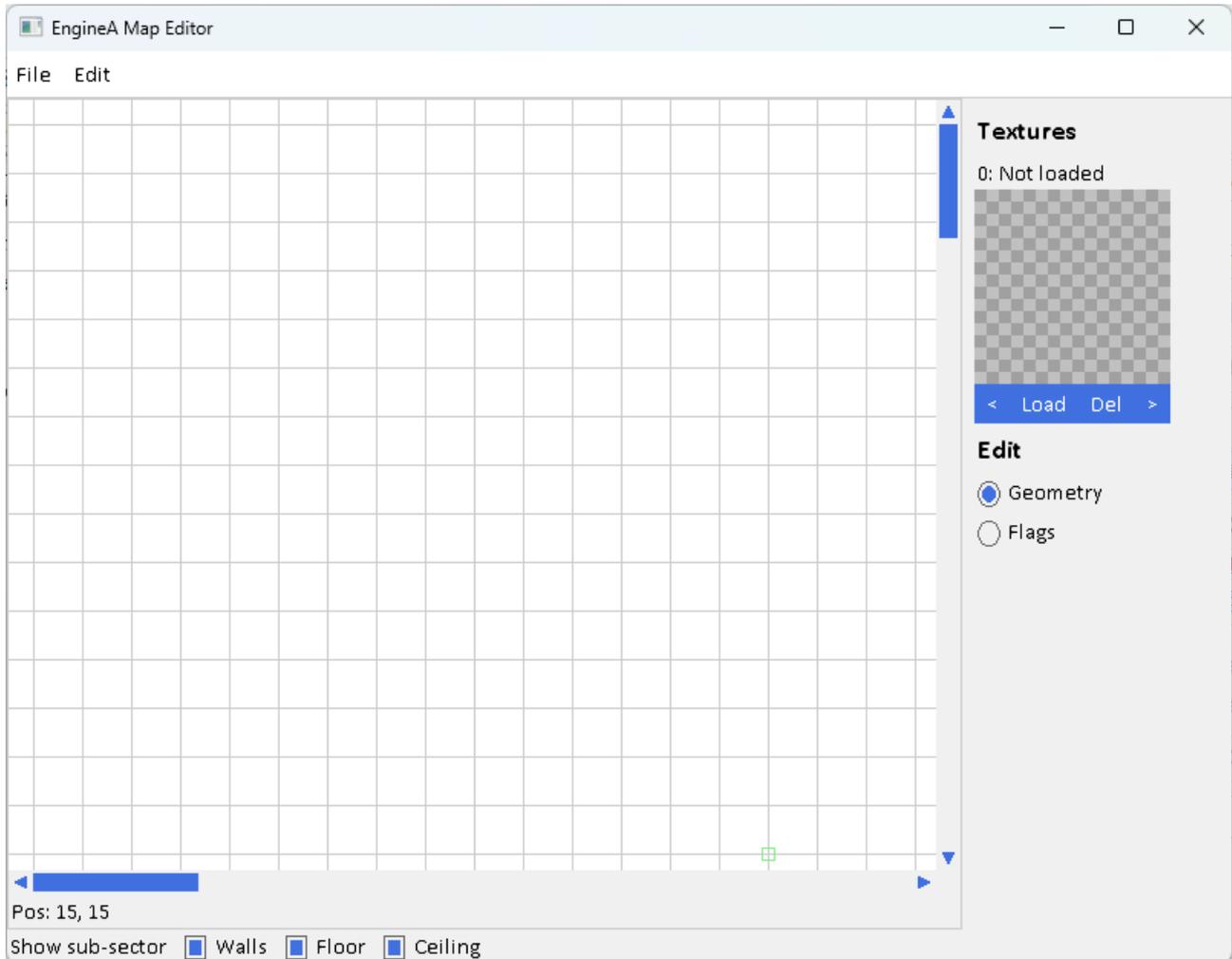
1 A single room

This example shows you how to:

- Create a map consisting of only one sector in the EngineA Editor (assets/map_1.json)
- Write the n7 code to load that map and let the player move around in (example_1.n7)

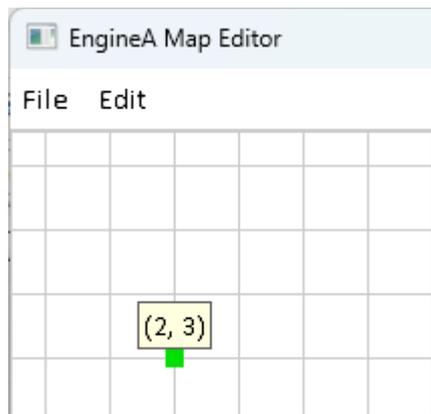
Create a folder where you can save your program and its assets (maps and images). I usually put all my assets in a sub-folder named "assets" next to the source code.

Launch the enginea editor (enginea_editor.exe) and this is what you'll see:

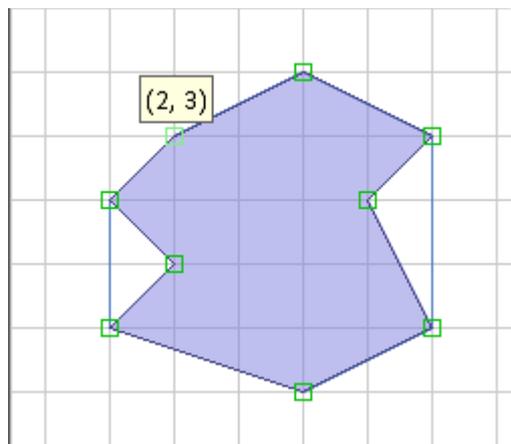


This is a topdown view of the level. The x axis (left to right) represents the x axis in the 3D world and the y axis (top to bottom) represents the z axis (actually the inverted z axis, but don't mind that for now). Just like in games such as Doom the map isn't really 3D (but creating height differences is still very possible).

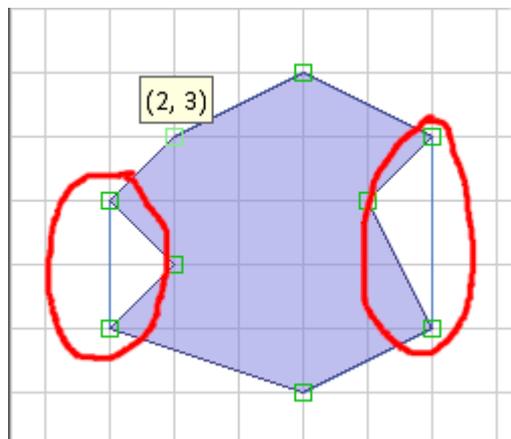
A sector is a closed polygon seen from above. Every line in the polygon is a wall and every vertex a corner. To create a new sector, click on an empty spot in the map and the sector's first vertex will appear as a green square:



Then keep adding points to the sector polygon by clicking. Close the polygon by clicking on the first point you added. A sector can be non-convex, but you shouldn't do silly things, like letting lines (walls) cross eachother – there's no error checking for that.



Now that's a fine room if I may say so! The currently selected sector (the one you just created) is always filled with a blue color. But what about those two extra blue lines?



They're there to show you the convex hull of the sector. You don't need to care about the convex hull right now, but it becomes very important when connecting sectors with portals.

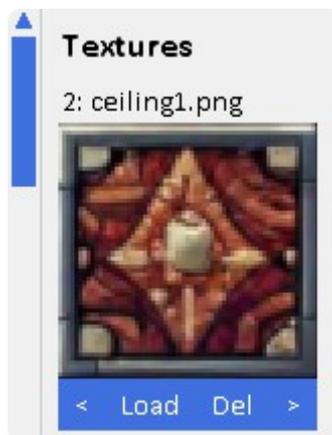
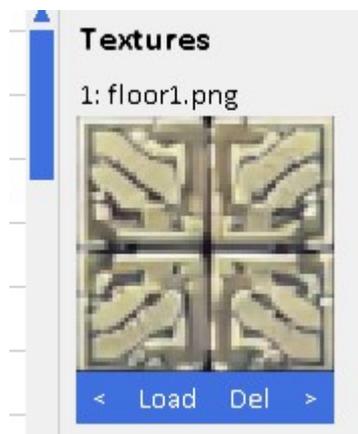
You can click on a vertex (unfilled green square) and drag it to another position if you need to.

Save your masterpiece by selecting "Save map as" in the "File" menu!

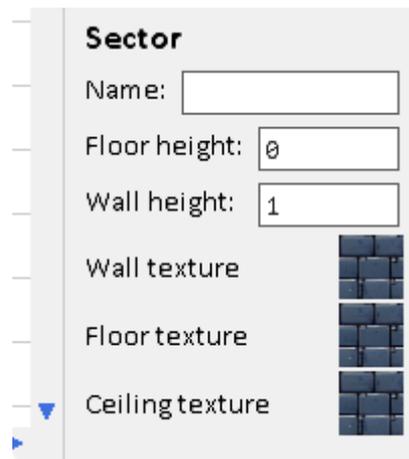
Now it's time to load some textures (images) for the walls, floor and ceiling of your sector. For unknown reasons, a map can use no more than 32 textures.



You can use the "<" and ">" buttons to change the texture index (0..31), and for each index you can click "Load" to load an image from file or "Del" to unload an image. In the image above I have loaded a texture for the walls. Then I increase the texture index and load another image for the floor and then one for the ceiling:



If you haven't clicked somewhere by mistake the sector should still be selected (blue). If it's not selected, just click on it again. If you click on the sector when it's already selected a green square will appear where you clicked, because the editor assumes that you want to create a sub-sector. If that happens, just press the Esc key. Sub-sectors will be covered in a later example.

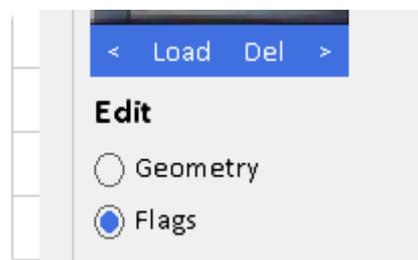


When a sector is selected, there are a couple of properties that you can set. You can give it a name by clicking on the text entry field next to "Name:". You can change its floor and wall height (covered later). Next to "Wall texture", "Floor texture" and "Ceiling texture" you can see small versions of the textures that are currently set for the walls, floor and ceiling. The textures default to index 0. Click on the image next to "Floor texture" and you can select any of the images that you previously loaded from a popup:

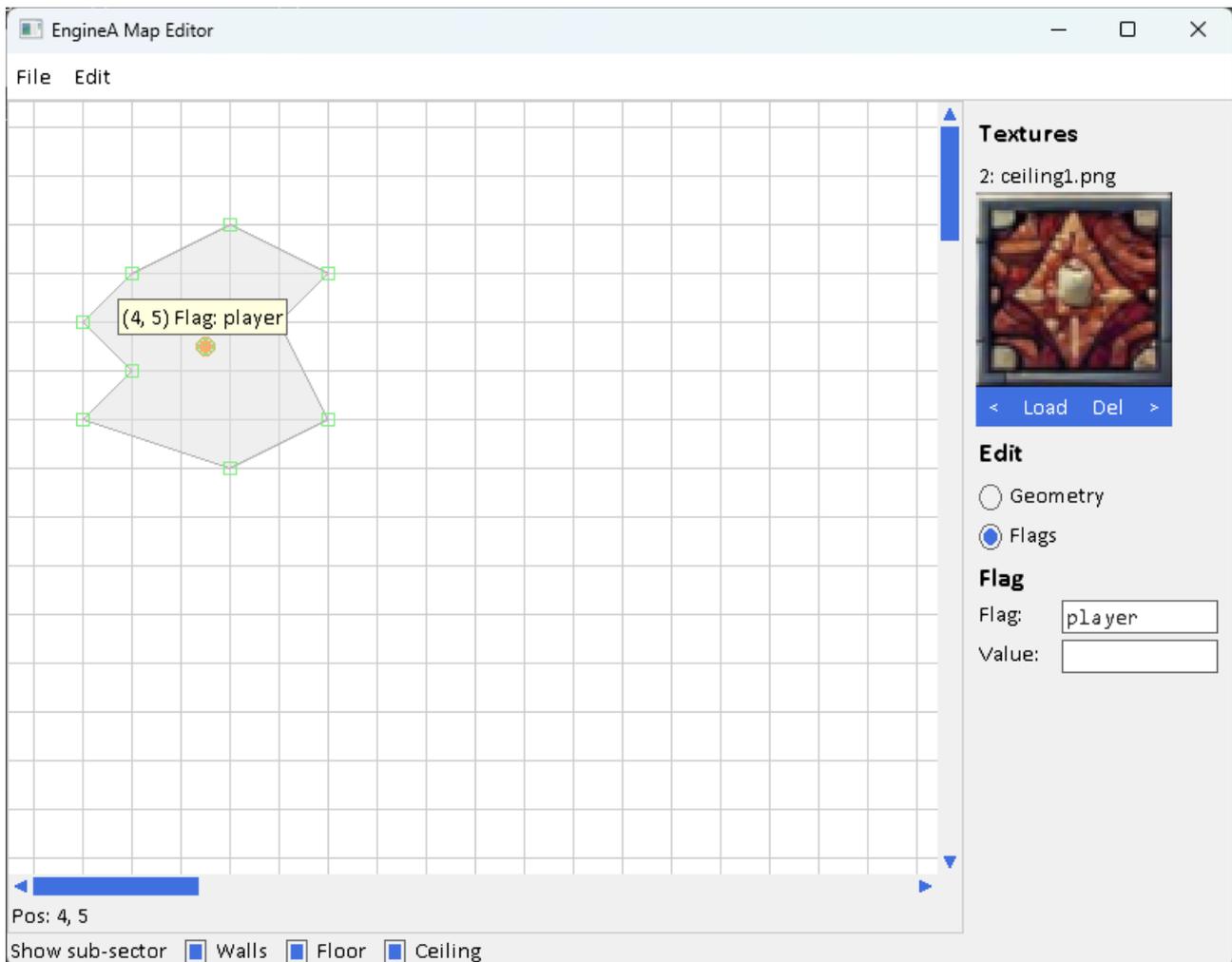


You can also set the texture used for an individual wall in a sector, but let's save that for later.

Only one thing is missing until we can launch NED and start writing code. We somehow need to mark the player's starting position on the map:



Change the radio button under "Edit" from "Geometry" to "Flags". Then click on the position in the sector where you want the player to start:



An unfilled circle will appear where you clicked. Next to "Flag:" under "Flag", type "player" in the text entry field and press return. Then save the map and close the program.

Now launch NED, create a new document and save it!

Start by including the EngineA library, create a window and turning automatic redraw off:

```
include "libs/enginea.n7"

set window "Example 1", 320, 240, false, 2
set redraw off
```

Call `EA_SetView(target_image, fov, z_min, z_max)` to set up the renderer:

```
EA_SetView(primary, rad(65), 0.1, 6)
```

`fov` is the vertical field of view in radians, and you can use `rad` to convert from degrees to radians. `z_min` is the near clip plane and `z_max` is the far clip plane. Anything rendered closer to the camera than 0.1 or farther than 6 units will be clipped. A "unit" is, simply put, a grid square in the EngineA editor. The larger your `z_max` value is, the more graphics need to be drawn.

Now use `EA_LoadMap(filename)` to load the map you created:

```
flags = EA_LoadMap("assets/map_1.json")
```

It returns an array with all the flags you added in the editor. If the function returns an unset variable, the map could not be loaded. So add an assert statement for that:

```
assert typeof(flags), "Map could not be loaded"
```

We just added one flag named "player", to tell us the player's starting position. Look for it in the array:

```
player = unset
foreach f in flags
    if f.flag = "player"
        player = EA_FpsPlayer()
        player.SetPos(f.x, f.floorY, f.z)
    endif
next
assert typeof(player), "No player flag found"
```

Each flag object in the array that EA_LoadMap returned contains these fields:

flag	flag name set in the editor
value	value set in the editor (we didn't set one, so it'll be an unset variable)
x	x coordinate of the flag
z	z coordinate of the flag
floorY	y coordinate of the floor at the flag's x and z position
ceilingY	y coordinate of the ceiling at the flag's x and z position

In the code above, when the "player" flag is found we call EA_FpsPlayer to create a new "game object" that contains lots of the functionality that you need for a player in a first person shooter game. Use the function SetPosition in this object to set the player's starting position to that of the flag. Later on you'll learn to configure the player object and create your own objects (enemies, bullets, decorations ...).

Next, add the player object to the game engine and set it to act as camera:

```
EA_AddObject(player)
EA_SetCamera(player)
```

We're almost done now. When we call EA_Run() the game loop starts and we lose control. That function won't return until we call EA_Stop(). But how are we supposed to call EA_Stop if EA_Run enters a loop? Well, we do that by setting up a callback function. Start by creating a function named Update (you can name it anything you want, of course):

```
function Update(dt)
    if keydown(KEY_ESCAPE, true) EA_Stop()
```

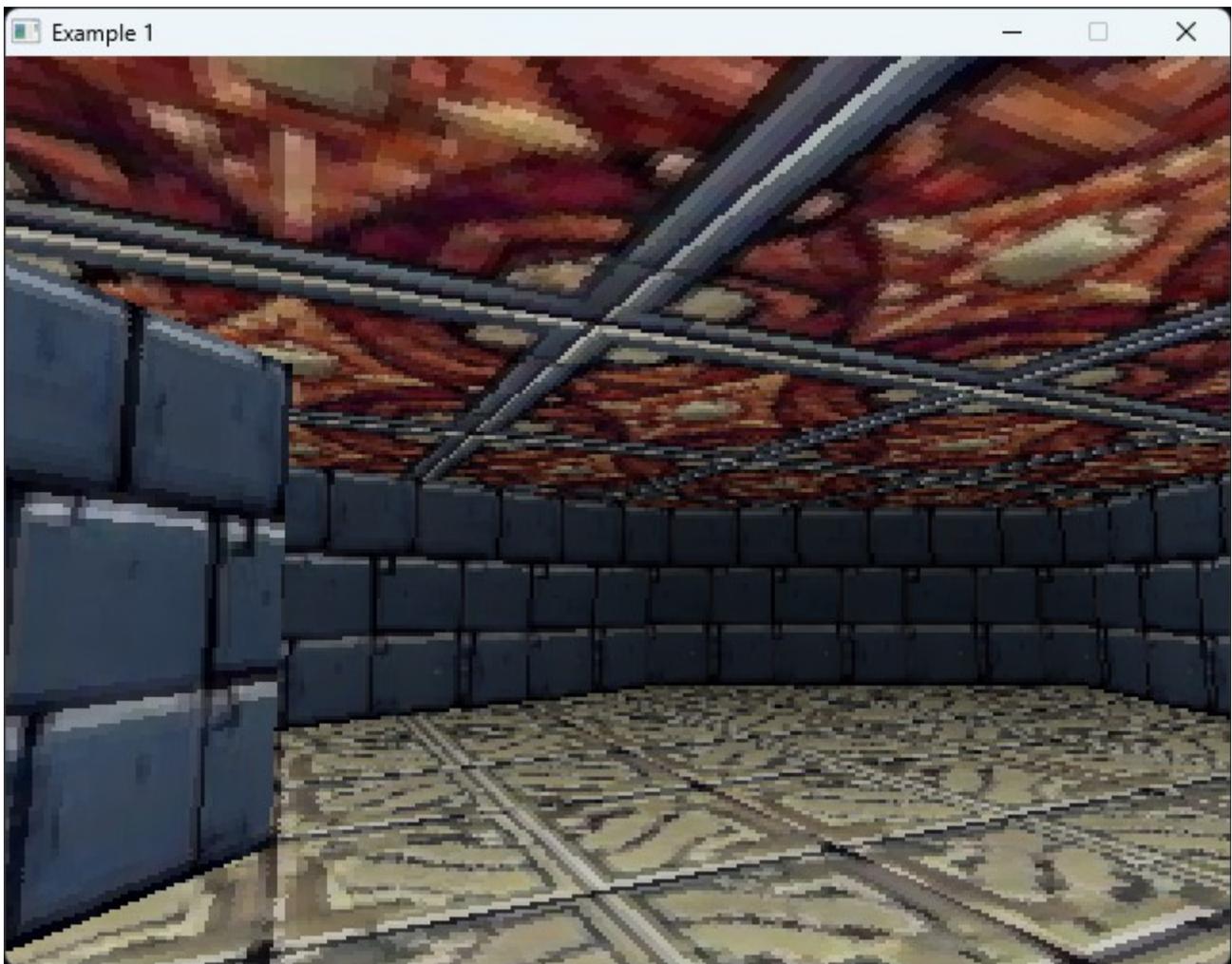
```
endfunc
```

Then use `EA_SetUpdateAction(update_func)` to set this function as a callback function that will be called once per frame.

```
EA_SetUpdateAction(Update)
```

When that is done, you add `EA_Run()` and compile and run your program.

```
EA_Run()
```



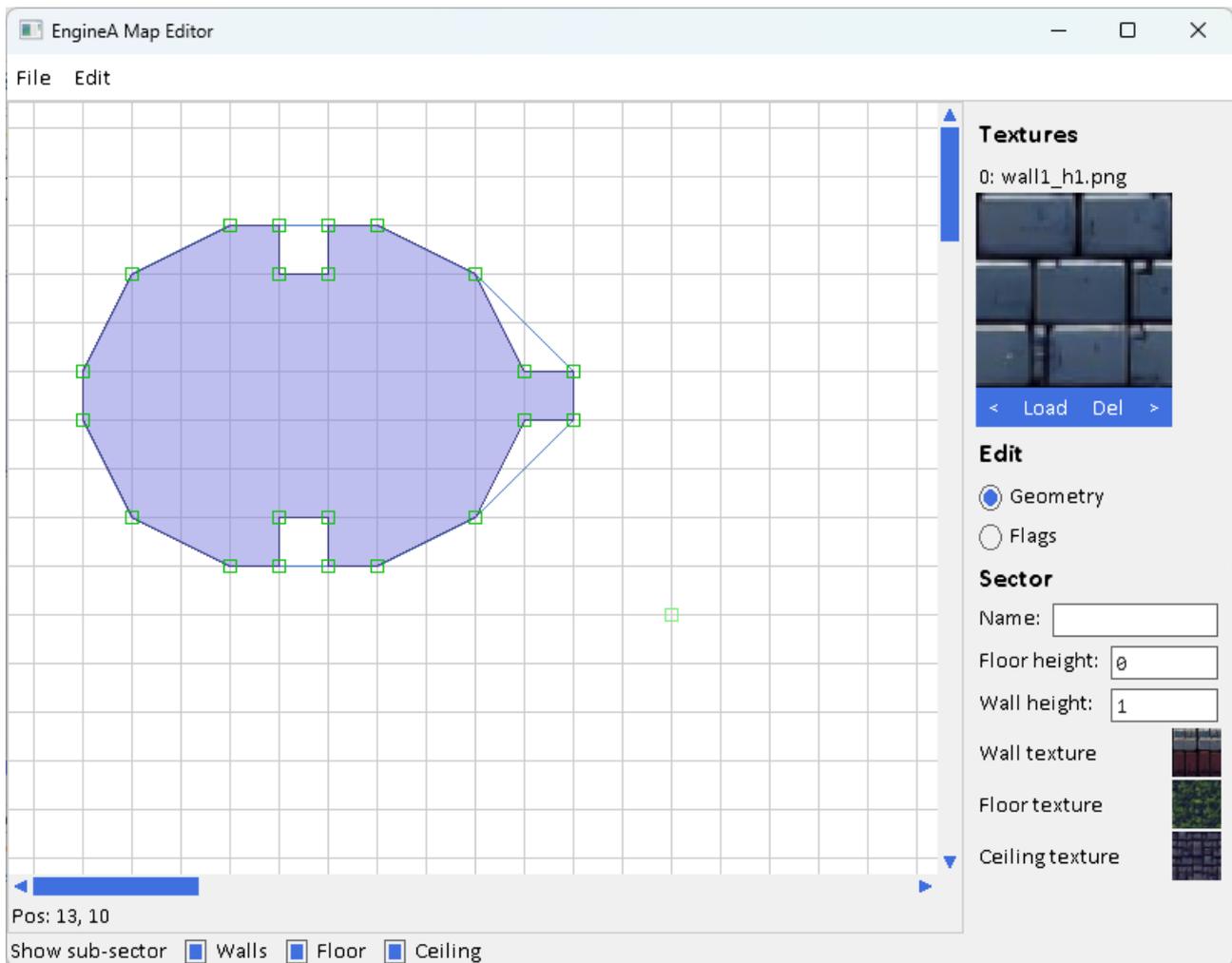
If you look at `example_1.n7` I also show you how to tweak the mouse sensitivity for the player and how to set up a fog effect.

2 Connecting sectors with portals

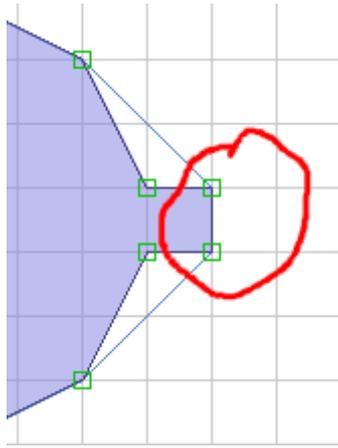
This example shows you how to:

- Create a map consisting of several sectors connected by portals (assets/map_2.json)
- Implement n7 code to open doors (example_2.n7)

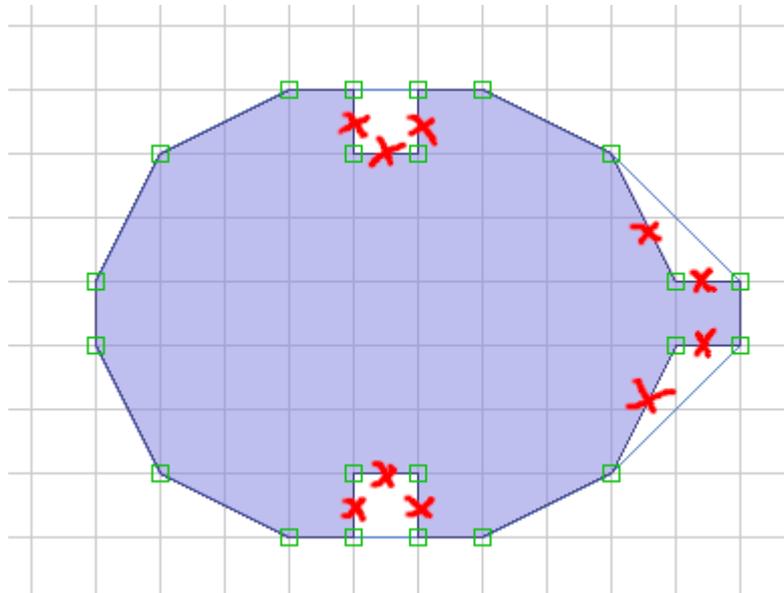
Create a new map in the EngineA Editor, load some textures and create a nice new sector somewhere and set up its wall, ceiling and floor textures:



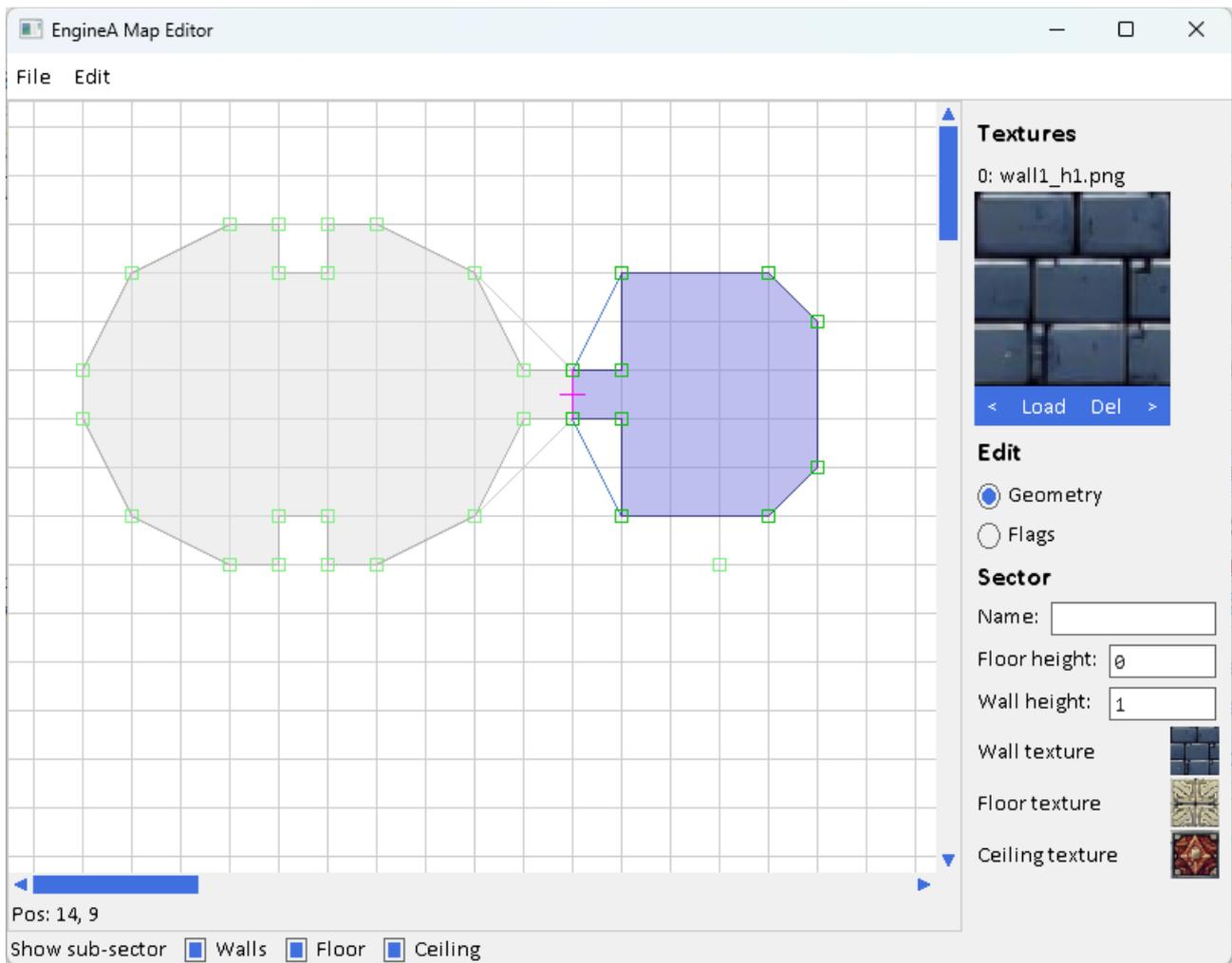
Let your sector have a wall that will serve as an opening to another sector. I will use the wall marked here:



The whole point of sectors and portals is to limit what needs to be rendered. So the wall that shall become a portal shouldn't be too long. The wall must also be part of the sector's convex hull. Remember the extra blue lines from the previous example? If you don't know what a convex hull is, those lines are there to help you see it. On this image, I've marked all the walls in the sector that can't become portals since they're not part of the convex hull:

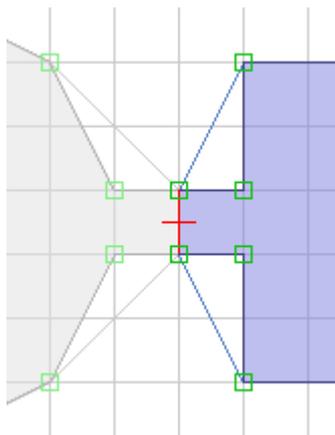


Now click somewhere outside of the sector to unselect it. Then draw a new sector and let it share one of its walls with the intended portal in your old sector:



When you close the polygon, the shared wall will become purple and marked with an extra line going through it. That means a portal, a doorway, has been created between the two sectors.

While you're at it, you can create a door, that can be opened and closed. With any of the two sectors selected, click on the portal. Its color will change to red:



And some settings for the wall and portal will appear under the sector properties:

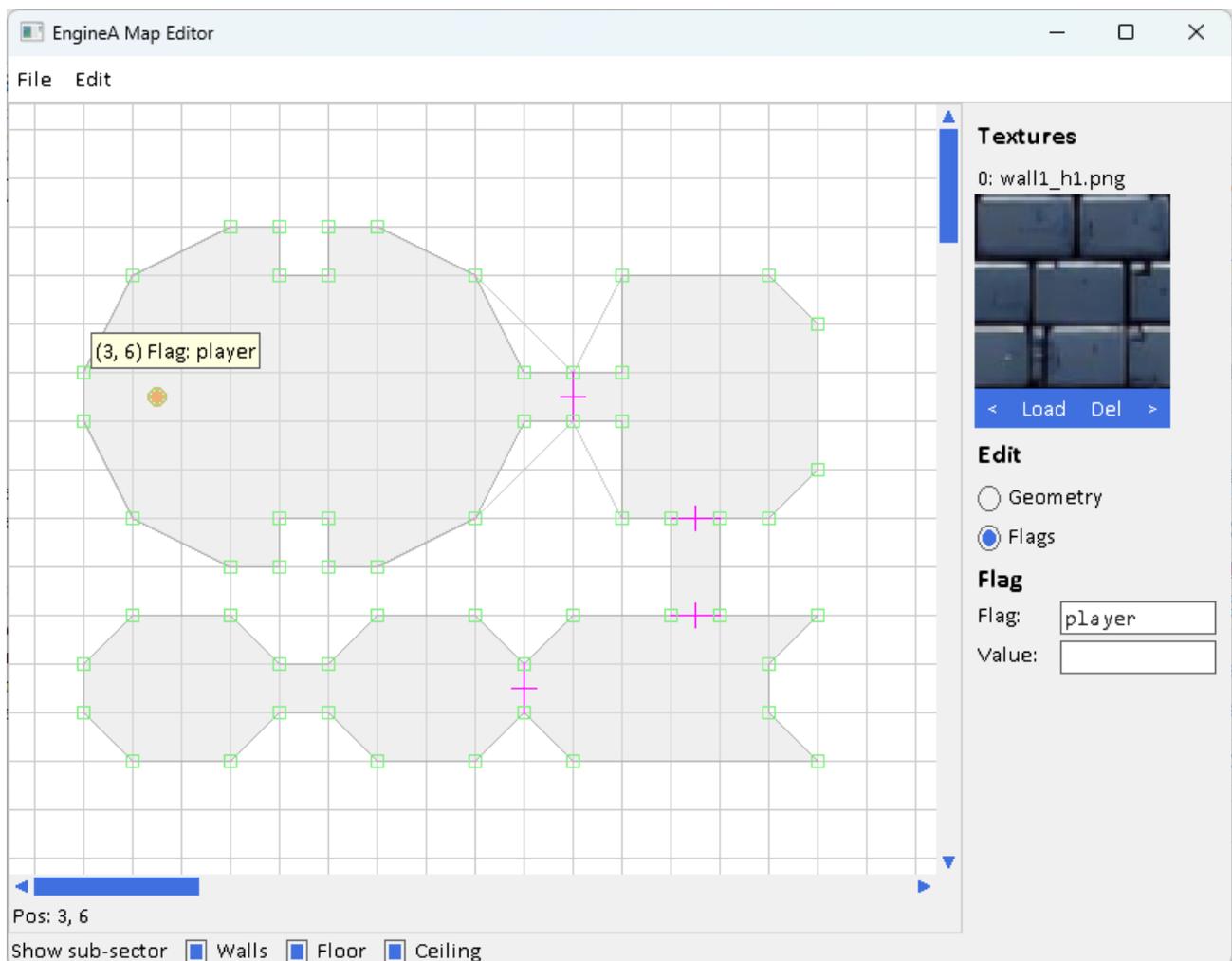


You can select any wall in a sector to set an individual texture for it by clicking on the small image next to "Texture" under "Wall". The option to set a "Door" texture only appears when you click on a portal.

Now, add some more sectors and connect them with the ones you've already created.

If you need to split a wall, add another vertex, click on the wall and select "Split wall" from the "Edit" menu! You can also cut copy and paste sectors by selecting them and using the "Edit" menu or the regular Ctrl+C, Ctrl+V and Ctrl+X key combinations. Portals will be created whenever two sectors share a wall.

Don't forget to add a flag for the player's starting position! This is how my map turned out:



After saving your map, start NED and copy the code from the previous example. Change the parameter to EA_LoadMap to match that of your new map and compile and run the program.

```
flags = EA_LoadMap("assets/map_2.json")
```

If you've added doors to your portals, you'll notice that there's no way to open them:



The object returned by `EA_FpsPlayer` can't open doors. Why not? Well, because most likely you will want to control which doors can be opened with or without keys and so on. So we'll have to write some code.

Remember the Update callback function that the game engine calls once every frame? Every object you create can be assigned a function that is called every frame. Somewhere after creating the player object with `EA_FpsPlayer`, assign a function named `Update` to it:

```
player.Update = function(dt)
endfunc
```

The `dt` parameter is the elapsed time in seconds since the function was called the last time. That goes for the program's update callback function too.

Now, what key should be used for opening doors? Let's go with the F key, next to WASD. I post some rather nasty code here and then try to explain what you need to know about it:

```
player.Update = function(dt)
    if keydown(KEY_F, true)
```

```

        res = .Facing()
        if res and res.type = EA_DOOR and res.dist < 1 then res.data.Open()
    endif
endfunc

```

Facing is a function that all game objects can use. ".Facing()" is short for "this.Facing()", in case you missed that in the release notes some n7 versions ago. Facing returns an object with lots of information about the closest wall, door or static object (a later topic) in front of the the calling object (the player in this case). If the player is looking at the ceiling or the floor, Facing currently returns an unset variable. So we start by checking if res has a value with "if res". Then we check the type field. If it's EA_DOOR, the player is facing a door. The dist field is the distance to the thing the player is facing. Since the player shouldn't be able to open a door from half across the room, we make sure that dist is less than 1 unit. If all the conditions are met we call the Open function on the data field of res. When the type field is EA_DOOR, the data field is a door object, and the door object contains the following functions:

Open()	opens the door, returns true if the door isn't already open or opening
Close()	closes the door, returns true if the door isn't already closed or closing
GetTexture()	returns the texture index of the door, set in the editor
X()	returns the center X coordinate of the door
Y()	returns the center Y coordinate of the door
Z()	returns the center Z coordinate of the door

If you run your program now, you should be able to open doors and visit all the rooms you created.

By default, doors open by moving up into the ceiling. You can use EA_SetDoorMode(mode) to change this behavior. The possible mode values are EA_SLIDE_UP, EA_SLIDE_DOWN and EA_SLIDE_SIDE.

But wouldn't it be nice if the doors would close automatically after a while? The renderer would certainly like it, since a sector behind a closed door needn't be drawn at all. To make that work, we create a visible array at the beginning of our program. It will contain all doors that are currently open.

```

visible vOpenDoors = []

```

Then we modify the player's Update function:

```

player.Update = function(dt)
    if keydown(KEY_F, true)
        res = .Facing()
        if res and res.type = EA_DOOR and res.dist < 1 and res.data.Open()
            vOpenDoors[sizeof(vOpenDoors)] = res.data
            res.data.timer = 3
        endif
    endif
endfunc

```

We add the door to the vOpenDoors array, and we add a field, timer, to the door object that we set to 3, which is the number of seconds the door will stay open. Now change the program's Update function so that it decreases the timer of all doors in the vOpenDoors array and close any door when

its timer reaches 0. When a door is closed, remove it from the array.

```
function Update(dt)
  if keydown(KEY_ESCAPE, true) EA_Stop()

  i = 0
  while i < sizeof(vOpenDoors)
    d = vOpenDoors[i]
    d.timer = d.timer - dt
    if d.timer <= 0
      d.Close()
      free key vOpenDoors, i
    else
      i = i + 1
    endif
  wend
endfunc
```

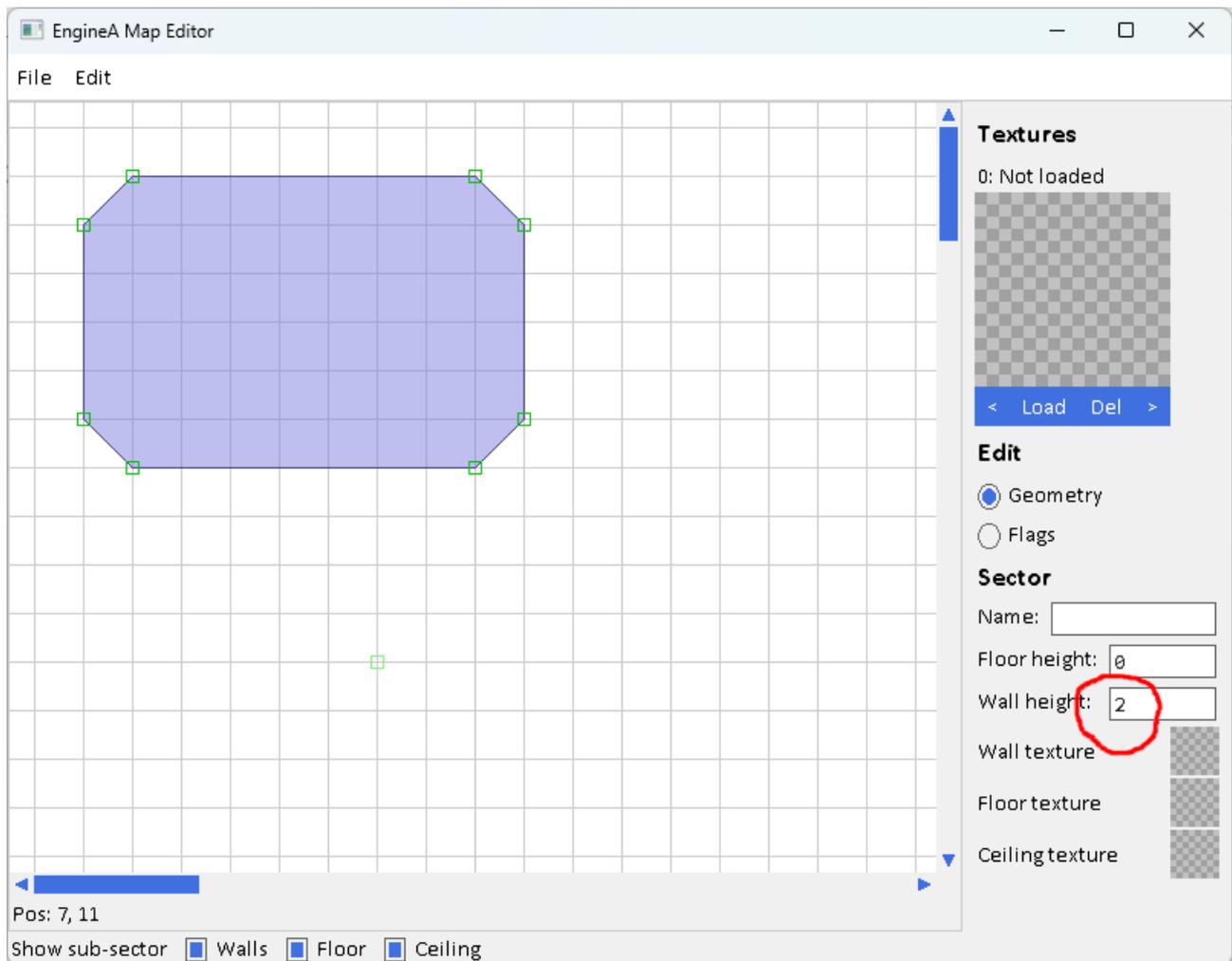
There, perfect! Run your program and the doors should automatically close. Don't forget to look at [example_2.n7!](#) In that version of the code, I adjust the player movement speed and some other things not mentioned here.

3 Sub-sectors and height differences

This example shows you how to:

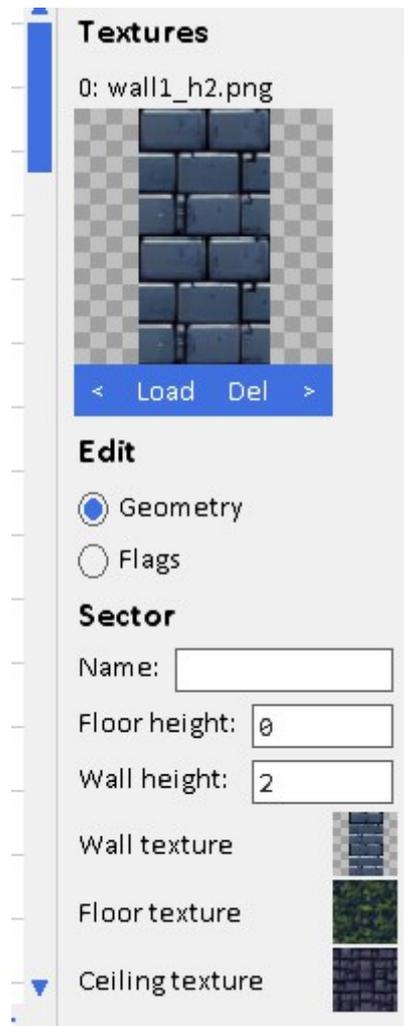
- Set the wall and floor heights for sectors and add sub-sectors to your sectors (assets/map_3.json)
- Write code to make the player handle heights well (example_3.n7)

Start the EngineA editor and create a rather large sector:

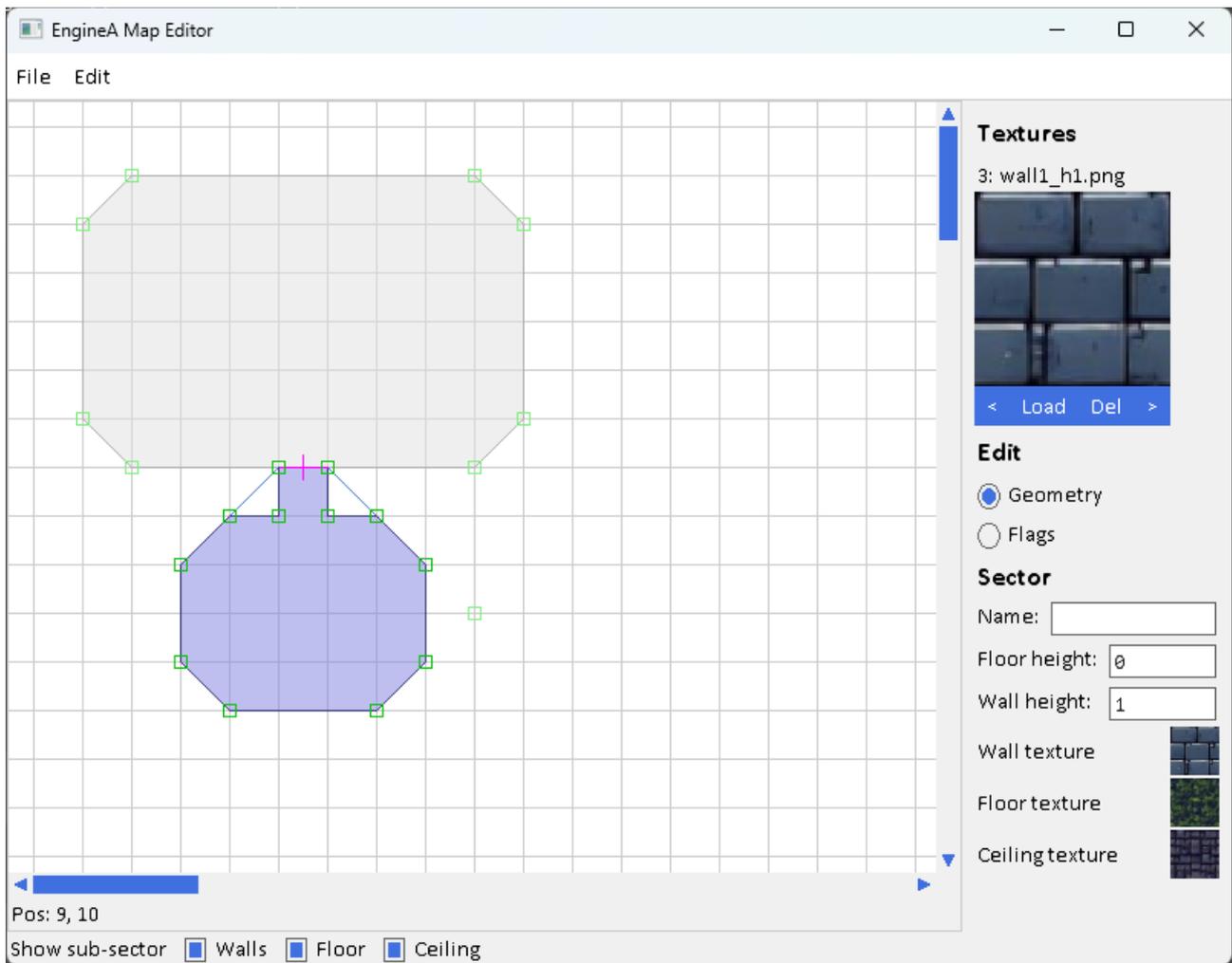


With the sector selected, change the "Wall height" value under "Sector" from 1 to 2. This will make the walls of this sector 2 units tall.

When the game engine builds a 3d mesh of a sector it repeats wall texture horizontally but not vertically. This means that the wall texture you use always represents the full height of a wall. Therefore I prefer to see the width of a texture as 1 size unit in the game. So if I want a room with a wall height of 2, I use a texture with the aspect ratio 1:2. Here I've loaded such a texture (and a bunch of others) and set it as wall texture for this sector:

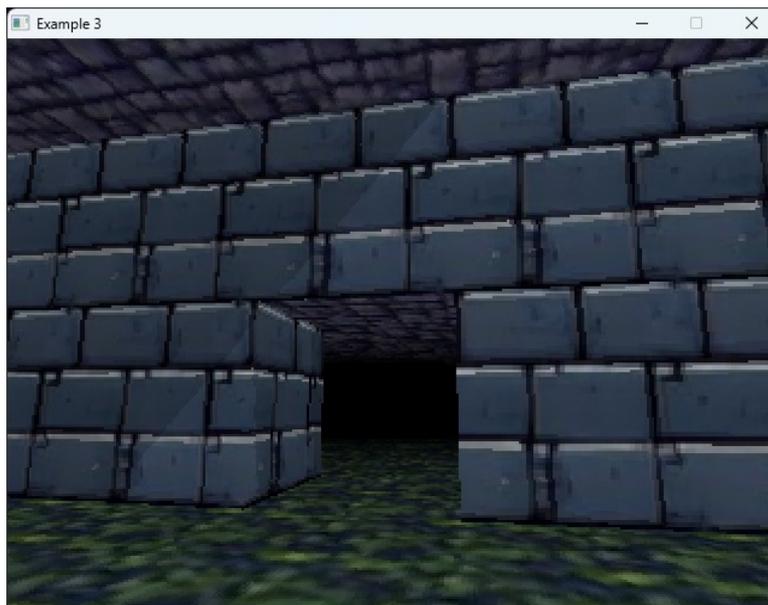


Add another sector and connect it to the first sector with a portal. Let the new sector's wall height be 1 unit. Following the logic about texture sizes that I just presented, the wall texture of this sector should have the aspect ratio 1:1:



Note that I had to add two vertices ("Edit" → "Split wall" twice) to the bottom wall of the first sector to form a portal to the second sector.

This is a rendering where the player is standing in the first sector, looking at the second sector through the portal:



The height of the portal depends on the height of the two sectors that it connects. Now let's try changing the floor height of the second sector. Set it to 0.25:

Sector

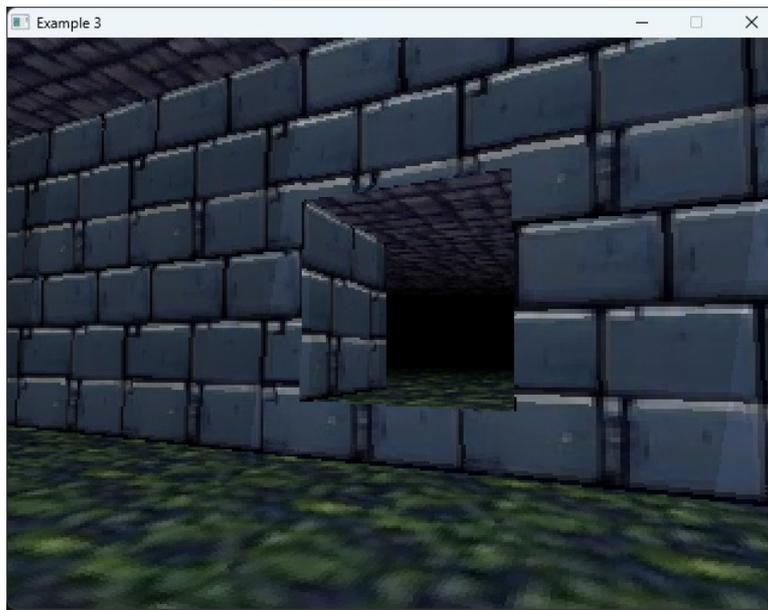
Name:

Floor height:

Wall height:

Wall texture 

Now the view from the first sector looks like this:

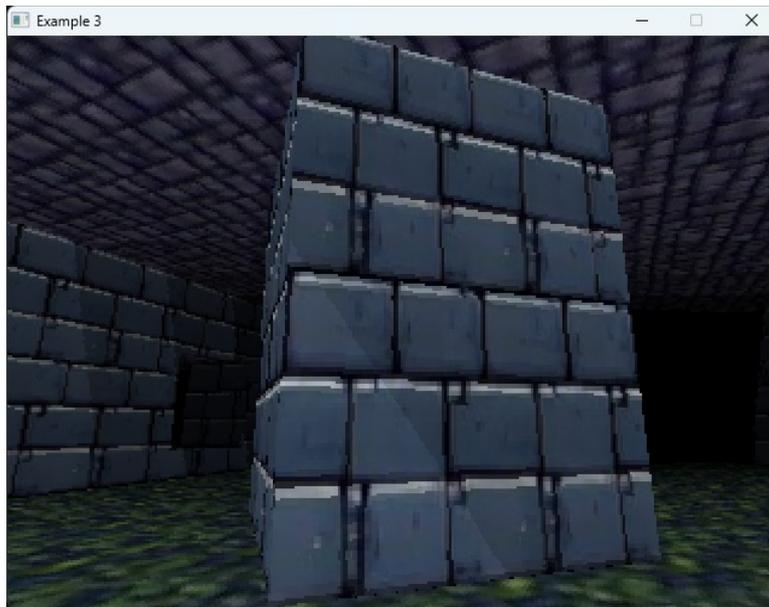


Fascinating! The height and vertical position of the portal is always adjusted to fit the floor and wall heights of its two sectors. And that is actually all there is to know about portals. So let's look at sub-sectors. They're fun.

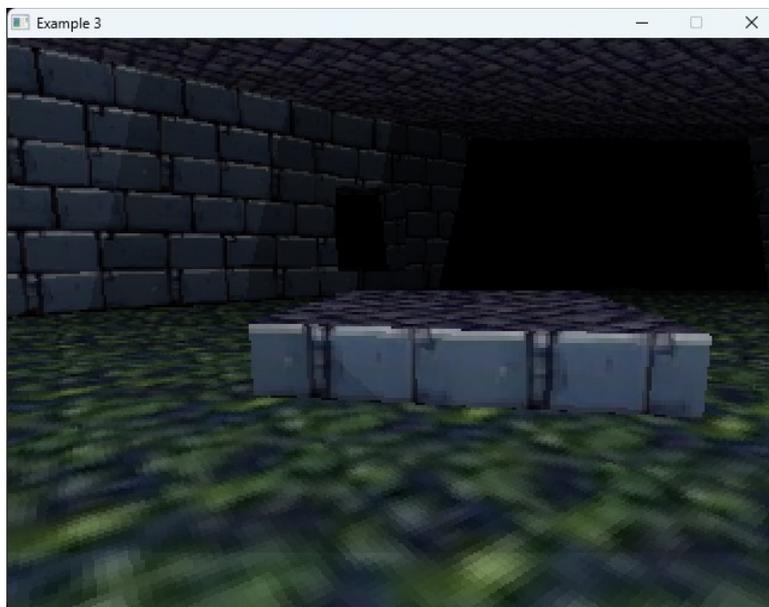
A sub-sector can be one of three things:

- Wall
- Floor elevation
- Ceiling submersion (I'm terribly sorry if "submersion" is the wrong word, I used google to translate the Swedish word "nedsäkning" and there were many options available ...)

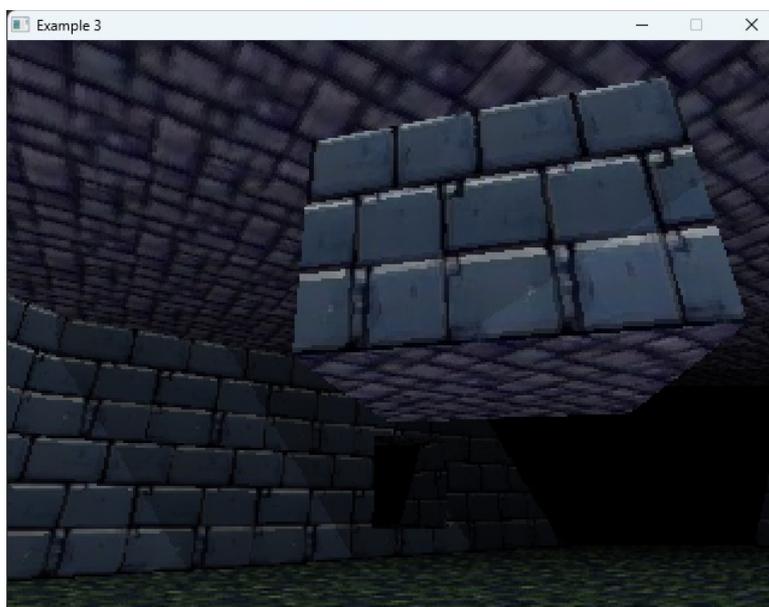
These images show the differences:



(Wall)

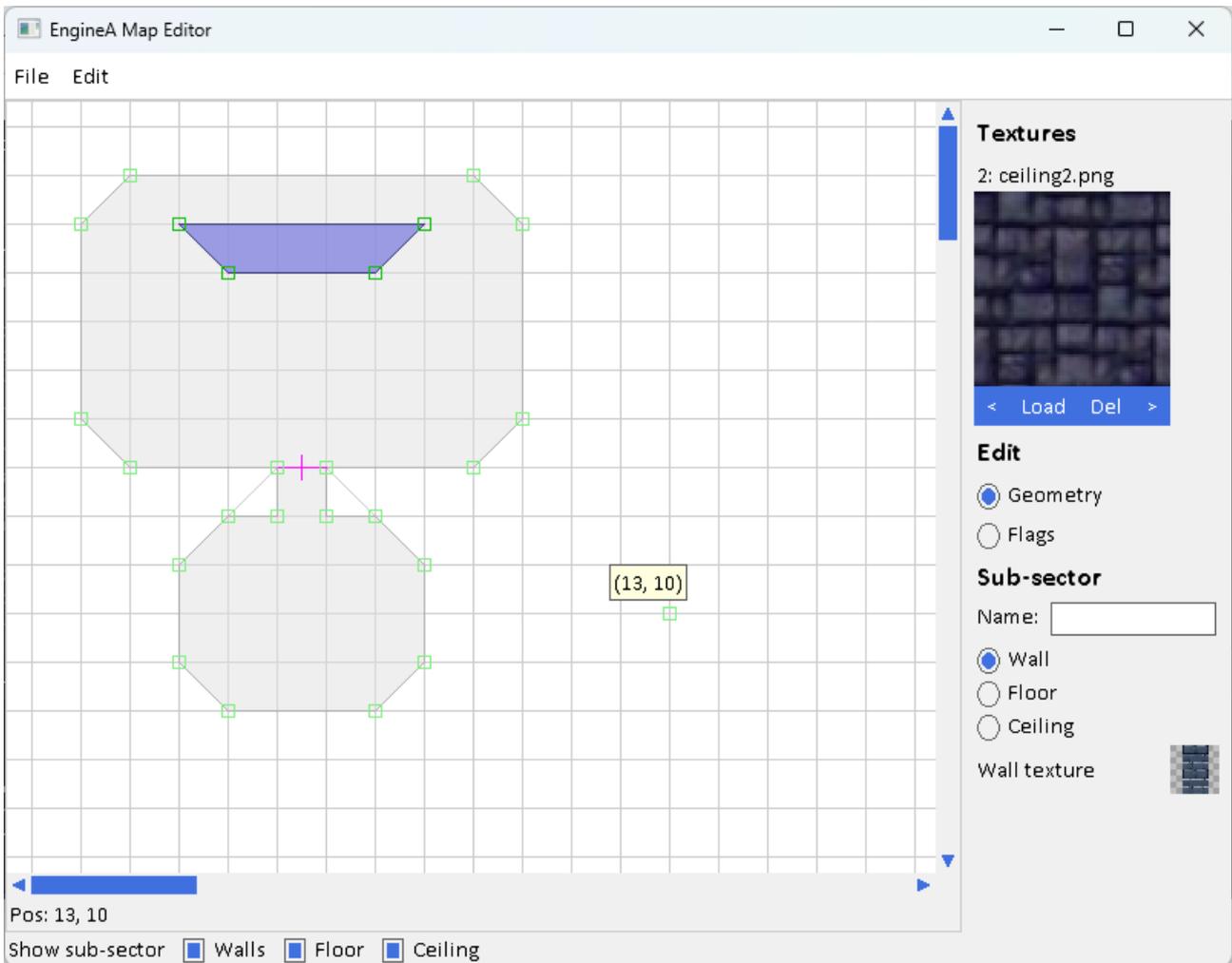


(Floor elevation)

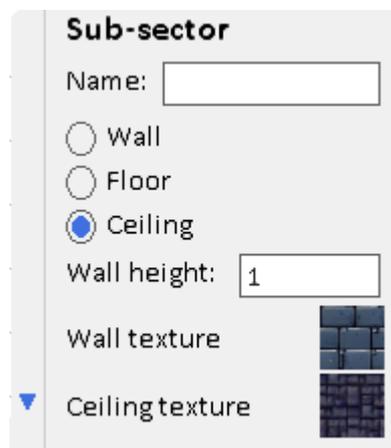


(Ceiling submersion)

Click anywhere inside a selected sector to start drawing the shape (polygon) of a new sub-sector. It works just like when you create a normal sector:



The sub-sector becomes the currently selected sector. With a sub-sector selected you can change its properties under "Sub-sector". Use the radio buttons to select one of the sub-sector types. "Floor" means floor elevation, and ceiling means ceiling submersion. You can set a wall texture for any type of sub-sector. But if you select "Floor" or "Ceiling" you will also be able to set a floor or ceiling texture:

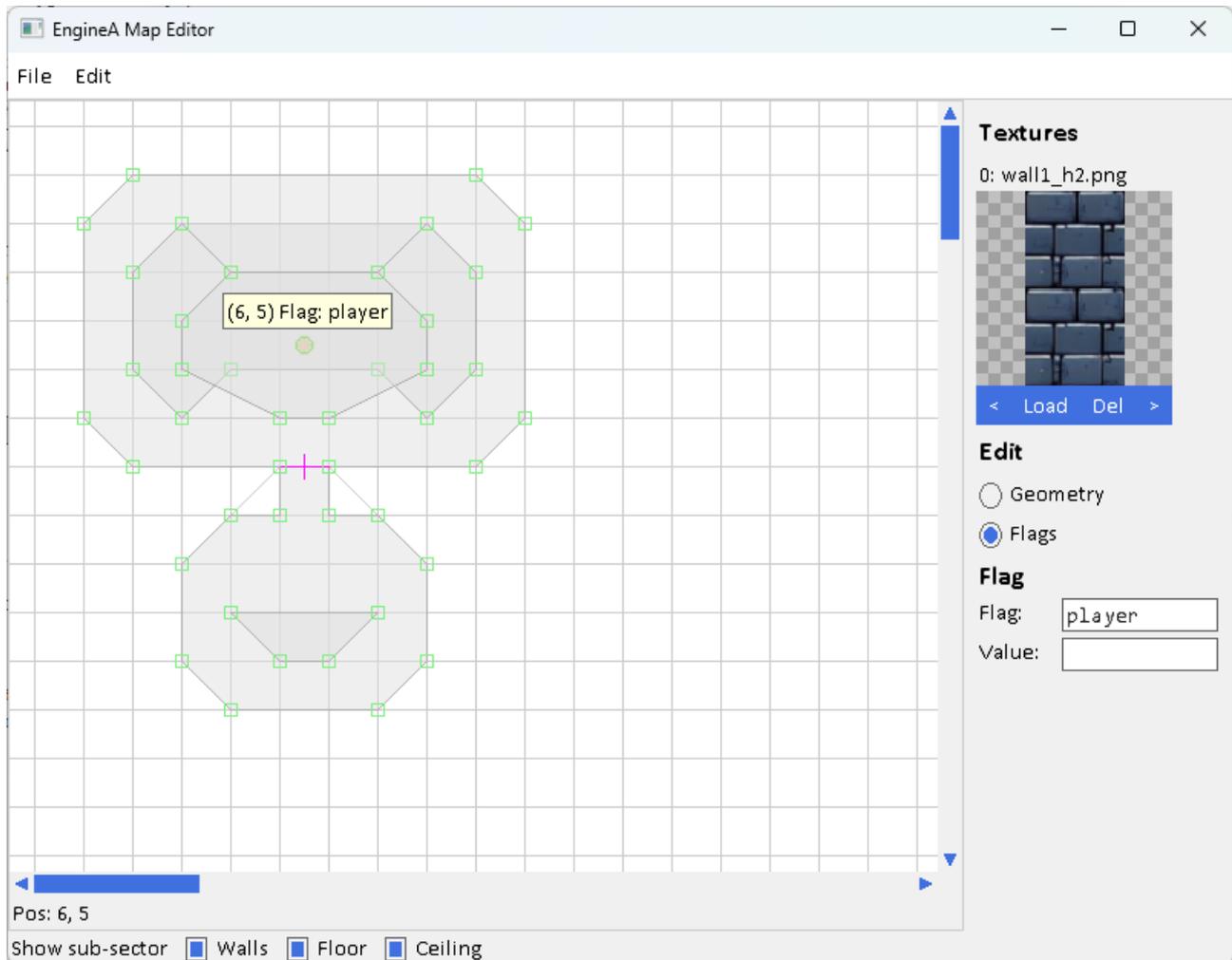


For "Floor" and "Ceiling" you should also enter a "Wall height" value, that represents the height of the elevation or submersion.

Add some sub-sectors to your sectors. You can toggle the visibility (in the editor) of the different sub-sector types using the check-boxes at the bottom of the editor:



Add a "player" flag somewhere on the map to mark the player's position and save your map. My map (assets/map_3.json) looks like this:



Launch NED and make a copy of the code you wrote for the previous example and change the filename in your EA_LoadMap call.

To make the player able to deal with heights, we have to add some lines of code. If you look at example_1.n7 and example_2.n7, you can see that I disabled jumping by setting the jump key to unset:

```
player.SetJumpKey(unset)
```

In example_3.n7 I remove this line, since I want the player to be able to jump. I also add these lines of code when setting up the player object:

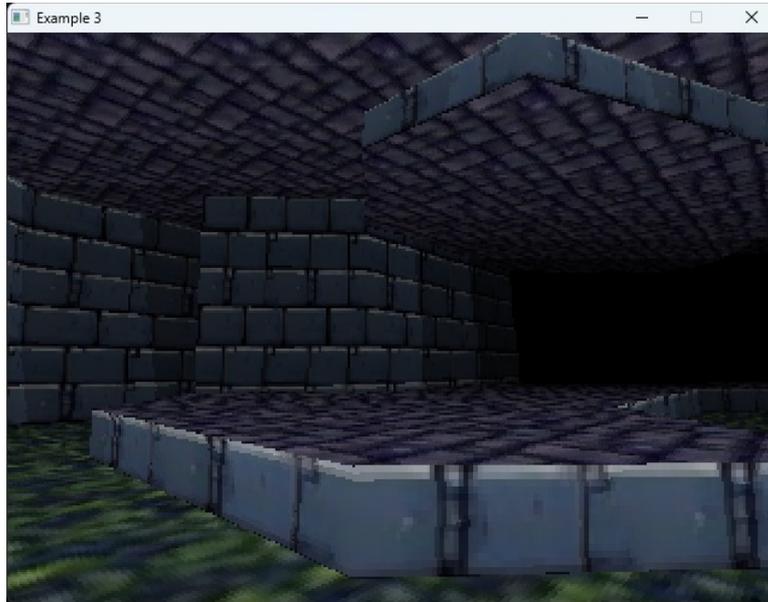
```
player.SetLeap(0.25)  
player.SetHeight(0.6)  
player.SetEye(0.5)
```

SetLeap makes the player able to automatically traverse heights ≤ 0.25 units tall. That is, the player won't need to jump to get up on them. You should keep this value low. On my map, there's a floor elevation where you can see the leap setting in action.

SetHeight sets the player's height, and SetEye sets the height of the players eyes, both default to 0.5.

Run your program and see what happens. Most likely you'll need to experiment a lot with sub-sectors to understand how they work in the game.

You can modify the player's "jump force" with SetJumpForce(force). The default value is 3. A higher value makes the player jump higher.



4 Simple non-moving sprites

In this example you'll learn to:

- Add "billboard" sprites to your level and check for collision with the player (assets/map_4.json and example_4.n7)

The engine makes a huge difference between static and non-static objects. The player returned by `EA_FpsPlayer` is a non-static object. And later on we'll use non-static objects for enemies, bullets and that kind of stuff.

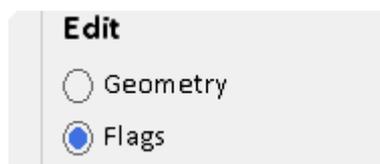
Static objects can't move once added. They can be assigned polygons for collision handling, so that they behave like obstacles to moving (non-static) objects. You may not be able to construct advanced geometry in the editor, but you can get around it by using static objects and meshes created in some modelling software. Static objects are also good for things that the player can pick up, which is what this step in the tutorial is all about. We will only use "billboard" sprites, no meshes yet.

Start by drawing or downloading some sprites that look like something a player would enjoy picking up. I used the Image Creator from Microsoft Designer to create these beauties:



I personally like it when the pixel density of my sprites matches that of the textures. That is, if a unit-sized texture is 64x64 pixels large, a sprite that is half a unit tall should use an image (texture) of height 32. But that's just me, a fan of really large pixels!

Start the map editor and re-use one of your old maps or create a new one. You can settle with a single rectangular sector if you want. There's no built in support for sprites or anything like that in the editor. Instead, switch to "Flags" under "Edit", just like you do when adding the "player" flag that determines where the player should start.



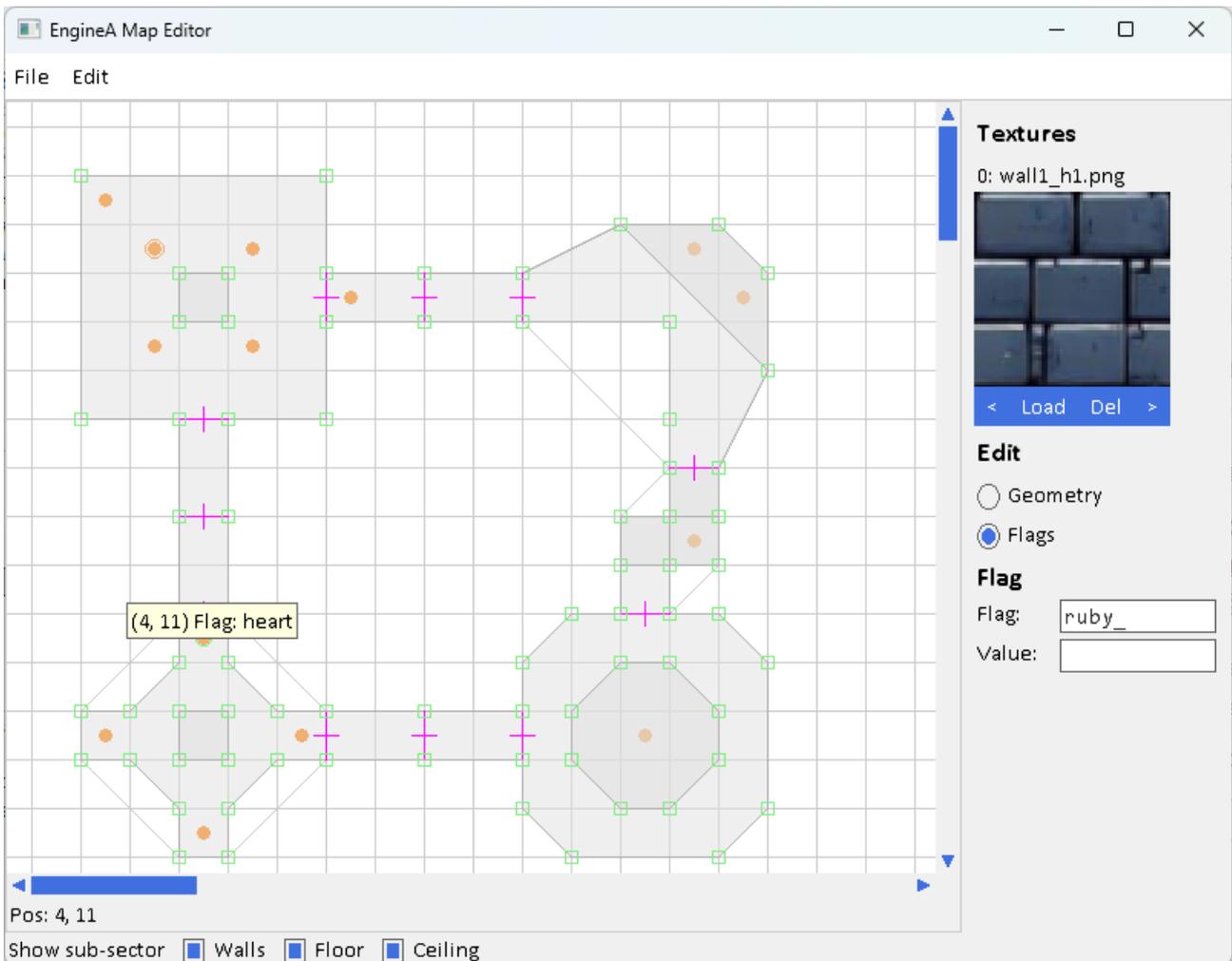
What kind of images did you decide to use? Mine are a ruby, diamond, heart and a treasure chest. Now, just as you do with the player, click on a map spot where you want to add one of your sprites. The spot will be marked with an unfilled circle (unfilled because it doesn't yet have a flag). Click on the text entry field next to "Flag" under "Flags" and enter the name of your item (it doesn't need to match the filename):

Flag

Flag:

Value:

Now, repeat that for every spot on your map where you want a sprite of that type. And if you've got more than one sprite image, add a couple of those too. My map is full of "ruby", "diamond", "heart" and "chest" flags. If you let the mouse hover over a flag on the map, information about it will appear in a tooltip:



When you're done adding flags for your sprites, save the map, shut down the editor and launch NED!

Now we must write the code for creating and adding static objects (sprites) when we find them in the flags array returned by `EA_LoadMap`. We also need to write code to check when the player touches any of the sprites.

Copy the source code from the previous step. Add code to load your sprite images into visible variables:

```
visible vRubyImage, vDiamondImage, vHeartImage, vChestImage
```

...

```

vRubyImage = loadimage("assets/ruby.png")
vDiamondImage = loadimage("assets/diamond.png")
vHeartImage = loadimage("assets/heart.png")
vChestImage = loadimage("assets/chest.png")

```

The code where you create and position the player looks something like this, right?

```

flags = EA_LoadMap("assets/map_3.json")
assert typeof(flags), "Map could not be loaded"
player = unset
foreach f in flags
    if f.flag = "player"
        player = EA_FpsPlayer()
        player.SetPos(f.x, f.floorY, f.z)
        player.SetLeap(0.25)
        player.SetHeight(0.6)
        player.SetEye(0.5)
        player.SetMoveSpeed(1.5)
    endif
next

```

Since we're now looking for other flags, use select instead of if, and add one case for each of your different flags:

```

foreach f in flags
    select f.flag
        case "player"
            ...
        case "ruby"
        case "diamond"
        case "heart"
        case "chest"
    endsel
next

```

Let's focus on the flag "ruby". To create a new static object, call EA_StaticObject()

```

case "ruby"
    ruby = EA_StaticObject()

```

Next, call the object's function SetSprite(image_id, cel, onlyYaw). image_id is the image we want to use, in my case vRubyImage. This image has no cels, so I use 0 for the cel parameter. If onlyYaw is true, the sprite will always be turned against the camera, but it will only be rotated around the y axis. If onlyYaw is false, the sprite will also be rotated around the x-axis and always face the camera completely.

```

case "ruby"
    ruby = EA_StaticObject()
    ruby.SetSprite(vRubyImage, 0, true)

```

Now we must set the size of the sprite. You do so by setting its height and a radius:

```

case "ruby"
    ruby = EA_StaticObject()
    ruby.SetSprite(vRubyImage, 0, true)
    ruby.SetHeight(height(vRubyImage)/64)
    ruby.SetRadius(0.5*width(vRubyImage)/64)

```

No one: "Whoa! Slow down, Marcus, what the fudge are you doing here?"

Marcus: As I told you, I prefer it when my sprites use the same pixel density as the wall (and floor and ceiling) textures. Therefore, I use the width and height of the sprite image to determine the sprite's size. My unit sized textures are 64x64 pixels large. So to calculate the height of a sprite, I divide its image height by 64. And I set the radius to half the width divided by 64.

If you don't want that kind of hocus pocus in your code, you can just say: "Nah, I want *my* ruby to be 0.4 units tall", and adjust its radius based on that and the aspect ratio of the image. Something like:

```

ruby.SetHeight(0.4)
ruby.SetRadius(0.4*0.5*width(vRubyImage)/height(vRubyImage))

```

We mustn't forget to set the sprites position using the object function SetPos(x, y, z). And then we add it to the engine with EA_AddStaticObject(object). Now my code looks like this:

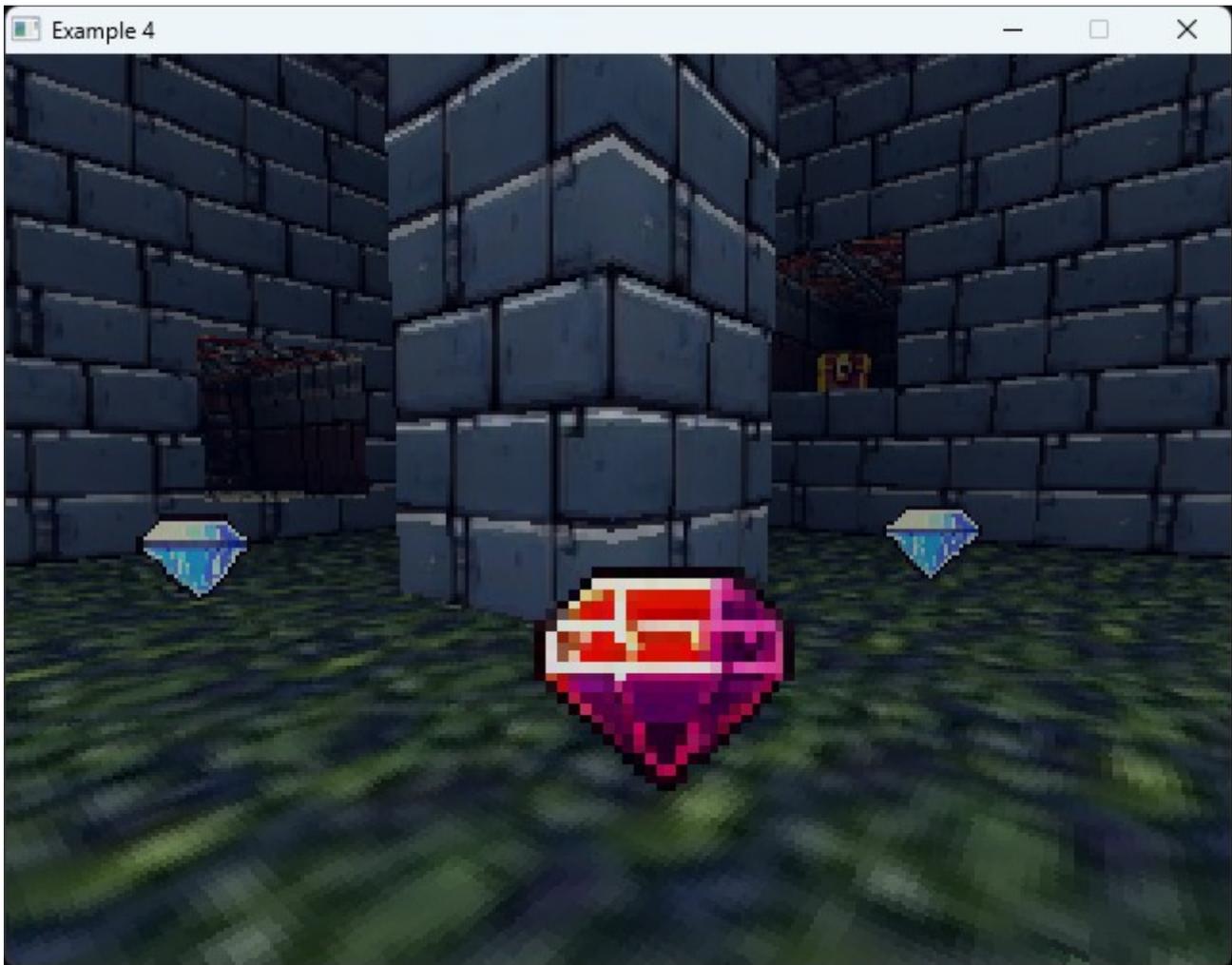
```

case "ruby"
    ruby = EA_StaticObject()
    ruby.SetSprite(vRubyImage, 0, true)
    ruby.SetHeight(height(vRubyImage)/64)
    ruby.SetRadius(0.5*width(vRubyImage)/64)
    ruby.SetPos(f.x, f.floorY, f.z)
    EA_AddStaticObject(ruby)

```

You can copy and modify the code you've written for your first sprite type to the rest. In example_4.n7 I've implemented a helper function that I call for each sprite type instead.

If you launch your program now, there should be some sprites on the ground where you put them in the editor:



But how do we make the player able to pick them up? That's quite easy. You need to add some code to your player object's Update function. You put some code there in an earlier step to make the player able to open doors.

```
player.Update = function(dt)
    ...

    objs = .SectorObjects()
    if objs and sizeof(objs)
        for i = 0 to sizeof(objs) - 1
            if .CollidesWith(objs[i])

                endif
            next
        endif
    endif
endfunc
```

You can use the object function `SectorObjects()` to get an array that contains all the objects in the sector that the object (the player) is currently located in. This array may possibly be unset, so check if it has a type with "if objs". Then check if it actually contains any objects with `sizeof`. Actually, it should always contain atleast ONE object, the object that is calling `SectorObjects`, but never mind that – I've already written the code! Loop through the array with a for statement.

The object function `CollidesWith(obj)` returns true if the current object (the player) intersects with the object `obj`. It simply compares the bounding cylinders of the two objects (based on the their

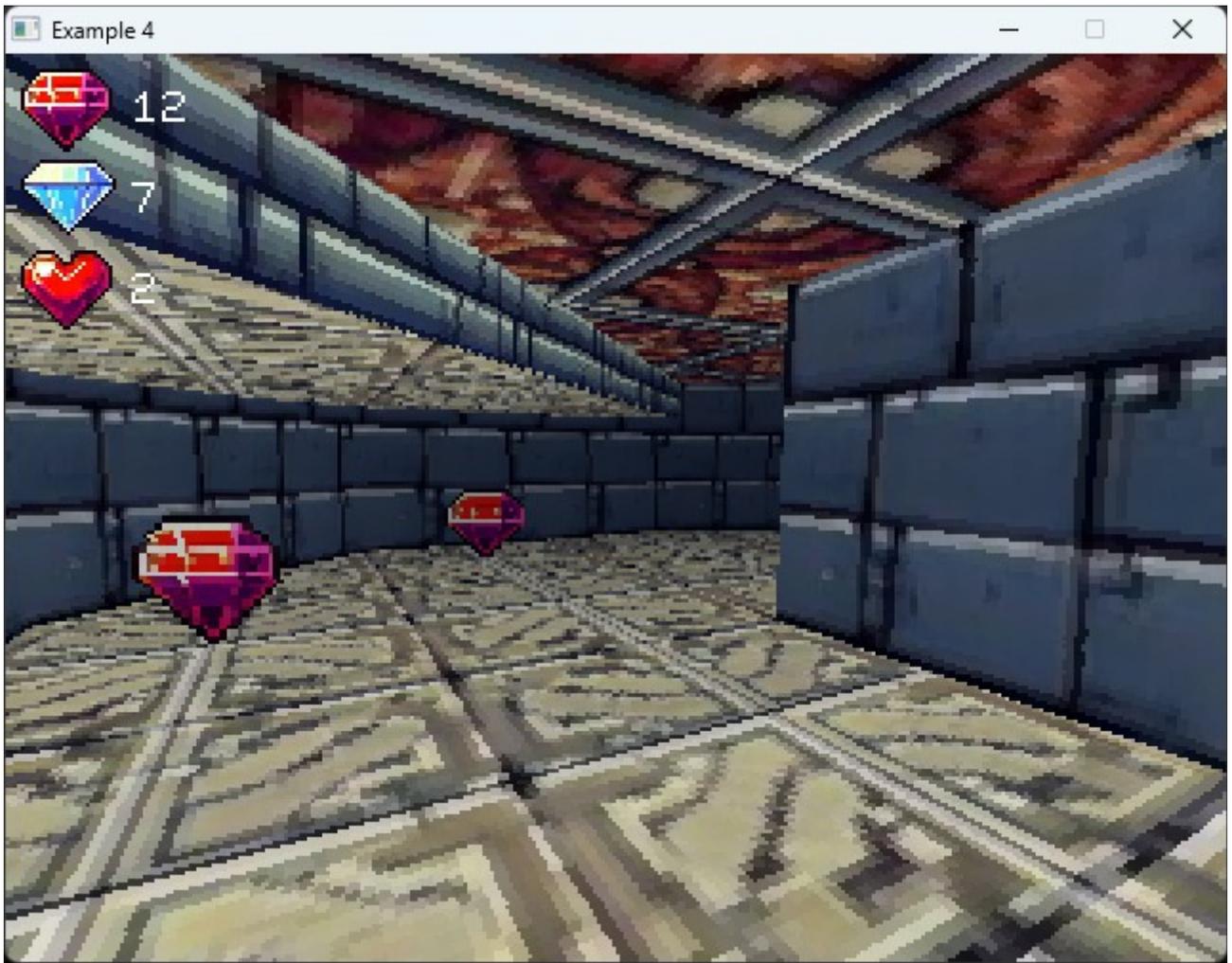
positions, radii and heights).

So, how do we determine what type of object `objs[i]` is? We are only interested in picking up (removing) the objects that are meant to be picked up, the ones we just created. One way is to check the sprite image of `objs[i]`, and that's just what we'll do. This is what the code looks like in `example_4.n7`:

```
objs = .SectorObjects()
if objs and sizeof(objs)
  for i = 0 to sizeof(objs) - 1
    if .CollidesWith(objs[i])
      select objs[i].Sprite()
        case vRubyImage
          vRubies = vRubies + 1
          EA_RemoveObject(objs[i])
        case vDiamondImage
          vDiamonds = vDiamonds + 1
          EA_RemoveObject(objs[i])
        case vHeartImage
          vHearts = vHearts + 1
          EA_RemoveObject(objs[i])
        case vChestImage
          vRubies = vRubies + 10
          vDiamonds = vDiamonds + 5
          vHearts = vHearts + 1
          EA_RemoveObject(objs[i])
      endsel
    endif
  next
endif
```

The object function `Sprite` returns the sprite image. We can compare it with the images that we assigned to our sprites. Use `EA_RemoveObject(obj)` to remove an object. As you can see, I'm incrementing some variables when the player picks up the different sprites. `vRubies`, `vDiamonds` and `vHearts` are visible variables that were initialized to 0.

We're done for now. But I suggest that you have a good look at `example_4.n7`! There I use `EA_SetDrawAction` to set a function that is called once per frame when the engine has rendered the 3D scene. In that callback function, you can draw whatever you want – in the example I use it to display the number of rubies, diamonds and hearts that the player has collected:



5 A very simple enemy sprite

This example shows you how to:

- Create an animated moving object (assets/map_5.json and example_5.n7)

Find or create a nice enemy sprite image! I, again, used the Image Creator from Microsoft Designer. Unless I'm missing something, it's very difficult to make it create sprites with animation frames. So I settled with a spooky pumpkin and animated it myself. This is my image grid, one row with four columns:



Yes, that's what pumpkins look like when they walk.

Create a new map or copy one of the old ones in the map editor. Add some flags for your enemies' starting positions. I use the flag "pumpkin". Save the map and launch NED.

Copy the code from the previous step! There's should be a new flag returned by EA_LoadMap for you to deal with. But start by loading your new sprite image into a visible variable:

```
visible vPumpkinImage
...
vPumpkinImage = loadimage("assets/pumpkin.png", 4, 1)
```

For cleaner code, I suggest that you put the construction of the new sprite in a function. This function will return a non-static game object (an object that can move), that we add with EA_AddObject. I named my constructor function CreatePumpkinEnemy:

```
flags = EA_LoadMap("assets/map_5.json")
...
foreach f in flags
  select f.flag
    ...
    case "pumpkin"
      o = CreatePumpkinEnemy(f.x, f.floorY, f.z)
      EA_AddObject(o)
    endsel
  next
```

Now, let's jump into the implementation of that function. Start by creating a game object using EA_Object(). Such an object contains lots of functionality:

```
function CreatePumpkinEnemy(x, y, z)
  enemy = EA_Object()
```

You will recognize these object function calls from the previous step:

```
enemy.SetSprite(vPumpkinImage, 0, true)
enemy.SetHeight(height(vPumpkinImage)/64)
enemy.SetRadius(0.5*width(vPumpkinImage)/64)
enemy.SetPos(x, y, z)
```

The functions work exactly as they do for static objects, and by now you know my thoughts about consistent pixel density. This next function call is new though:

```
enemy.SetYaw(rad(rnd(360)))
```

Here I give the enemy a random yaw angle. We will use this angle for the enemy's movement. There's also SetPitch to set a pitch angle, but we don't need that for this creature. If you're not familiar with yaw and pitch (and roll) angles, I must advise you to look it up. Basically, yaw is rotation around the y (vertical) axis and pitch is rotation around the x axis after applying the yaw rotation. To put it short, we give the enemy a random direction in the xz plane (top down view).

Since I'm using an image with four animation frames, a walk cycle, I then add a cel field to the enemy object:

```
enemy.cel = 0
```

Just like the player or any object, static or non-static, you can assign an Update function to the enemy. It will be called once per frame and gets the delta time in seconds as parameter:

```
enemy.Update = function(dt)
    .cel = (.cel + dt*8)%4
    .SetCel(int(.cel))
```

The first thing I do in the function is to update the cel field of the object. I use %4 to keep the value in the valid range of cels. Each second, cel will increment by 8 in total. That means that the walk animation will play twice every second. The object function SetCel(cel) sets the cel that the engine will use for the sprite.

Now it's time to move the enemy. I want the enemy to keep walking in its current direction until it hits a wall. Then it should get a new, random, direction. Let's start by moving him with the object function Move(dx, dy, dz, leap). It tries to move the object and returns lots of information about how it went:

```
res = .Move(.DX()*0.75*dt, 1*dt, .DZ()*0.75*dt, 0.25)
```

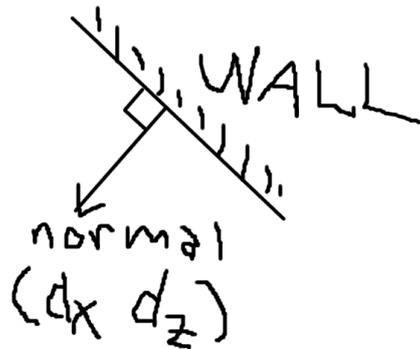
Whoa, that looks super weird, right? No? Oh. We're using SetYaw to set the objects direction (that's not mandatory, you could skip the yaw thing and manage the angles and directions yourself). When using SetYaw/SetPitch, you can get the corresponding direction using DX(), DY() and DZ(). We only need to use DX() and DZ() now. The dx and dz parameters DX()*0.75*dt and DZ()*0.75*dt means that the enemy will move 0.75 units per second. So what about the dy value, 1*dt? That's just a very stupid way of managing gravity. When the enemy is not on ground, he will move down 1 unit per second. The fourth parameter, leap, does the same thing as SetLeap does for the player;

internally, the player object calls Move with the leap value that you set. So we use the same leap value for the enemy that we do for the player, 0.25.

As I said, Move returns lots of information about what happened when the object moved. You can get information about what kind of wall or static object the object bumped into, if it hit the ground or the ceiling etc. But for this stupid enemy, we just need to know if it hit a wall. If it did, res.w will be true:

```
res = .Move(.DX()*0.75*dt, 1*dt, .DZ()*0.75*dt, 0.25)
if res.w
    .SetYaw(rad(deg(atan2(res.dx, res.dz)) - 90 + rnd()*180))
endif
```

Lots of code, again. You know what SetYaw does. res.dx and res.dz is the normal of the wall that the object collided with. The normal is a vector perpendicular to the wall surface. Look at this image (I drew it myself) that shows a wall and its normal from a top down perspective:

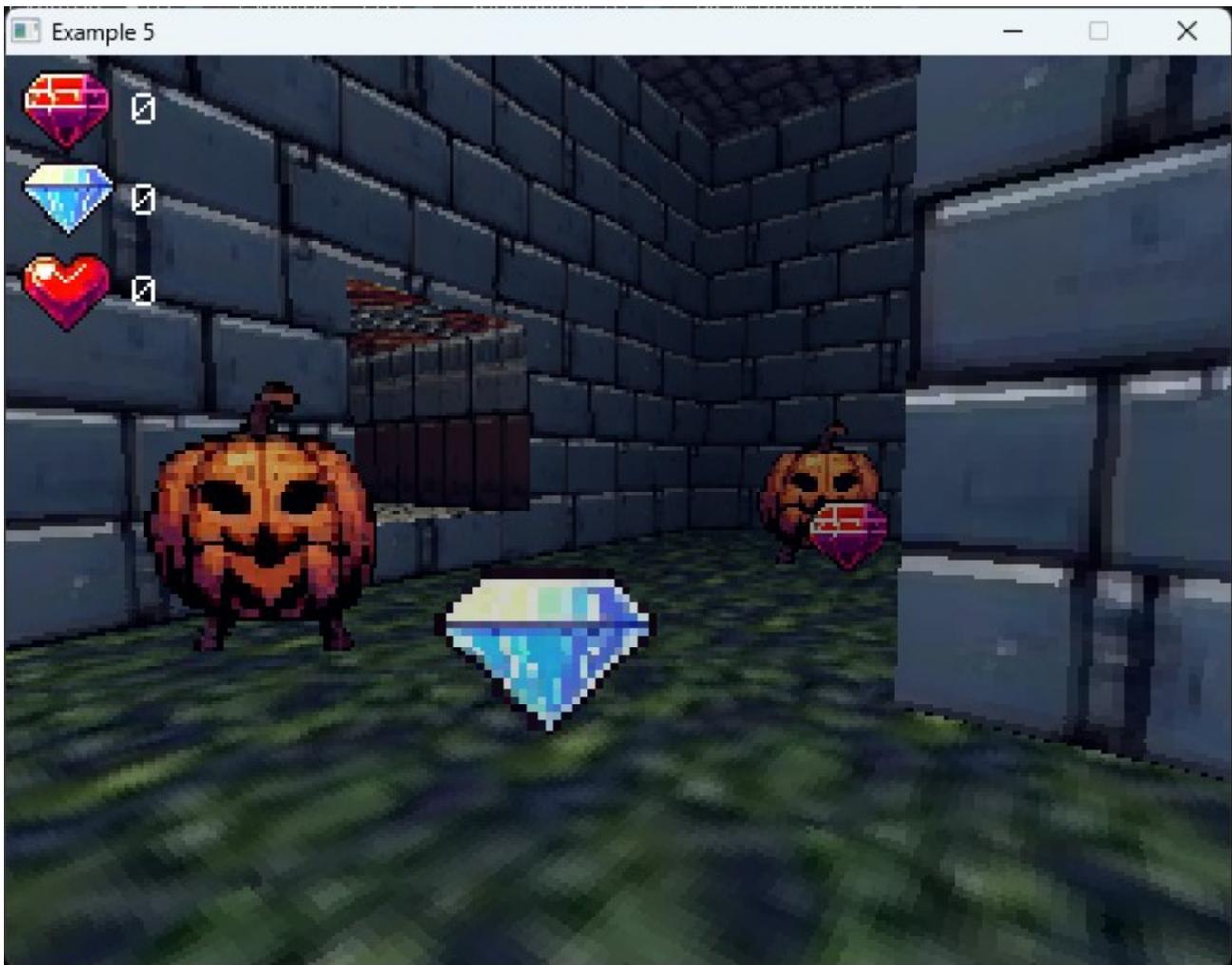


atan2(res.dx, res.dz) returns the angle (in radians) that corresponds to the normal. We want the enemy to be stupid, but not quite so stupid that he runs straight into the wall again after hitting it. So I use the angle returned by atan2, subtract 90 degrees and add 0..180 degrees by random. That way, the enemy will always walk away from the wall it just hit, but still in a random direction. If this feels like overkill, you can just skip the normal thingie and use SetYaw(rnd(360)) instead, but it's good if you try to understand what I'm doing here :)

Alright, that's the entire update function. Now my CreatePumpkinEnemy function looks like this:

```
function CreatePumpkinEnemy(x, y, z)
    enemy = EA_Object()
    enemy.SetSprite(vPumpkinImage, 0, true)
    enemy.SetHeight(height(vPumpkinImage)/64)
    enemy.SetRadius(0.5*width(vPumpkinImage)/64)
    enemy.SetPos(x, y, z)
    enemy.SetYaw(rad(rnd(360)))
    enemy.cel = 0
    enemy.Update = function(dt)
        .cel = (.cel + dt*8)%4
        .SetCel(int(.cel))
        res = .Move(.DX()*0.75*dt, 1*dt, .DZ()*0.75*dt, 0.25)
        if res.w
            .SetYaw(rad(deg(atan2(res.dx, res.dz)) - 90 + rnd()*180))
        endif
    endfunc
    return enemy
endfunc
```

That's not an awful lot of code for an enemy, right? When I run my program (example_5.n7) this is the horror that I see:



6 Boop the pumpkins

In this example you won't really learn anything new at all. We'll clean up the code a bit and make the player able to shoot bullets that push the enemies around (without hurting them). You can copy your map and code from the previous step.

Cleaning up the code

First of all we want a nicer way of identifying our objects (bonuses, enemies, bullets and so on) than looking at what sprite image they use (`GetSprite()`). Looking at the sprite image would get messy if some sprites change their images as part of animation or state changes. So add a unique constant for every type of game object that you create. My list looks like this (`PLAYER_BULLET_ID` will soon make sense):

```
constant PLAYER_ID           = 1
constant RUBY_ID             = 2
constant DIAMOND_ID         = 3
constant HEART_ID           = 4
constant CHEST_ID           = 5
constant PUMPKIN_ID        = 6
constant PLAYER_BULLET_ID  = 7
```

In the function where you create your enemy, the pumpkin in my case, assign the proper id to a field named `id`. This is how my `CreatePumpkinEnemy` function starts now:

```
function CreatePumpkinEnemy(x, y, z)
    enemy = EA_Object()
    enemy.id = PUMPKIN_ID
    ...
```

Where and how do you create the objects that the player can pick up? Are you doing it in the flag loop, or have you followed the example code and created a function for it? Wherever you call `EA_StaticObject`, assign the right id to the `id` field. I modified my `CreateItem` function:

```
function CreateItem(id, img, x, y, z)
    item = EA_StaticObject()
    item.id = id
    ...
```

, and added the proper id as first parameter when I look through the flags array:

```
foreach f in flags
    select f.flag
        ...
        case "ruby"
            o = CreateItem(RUBY_ID, vRubyImage, f.x, f.floorY, f.z)
            EA_AddStaticObject(o)
        case "diamond"
            o = CreateItem(DIAMOND_ID, vDiamondImage, f.x, f.floorY, f.z)
            EA_AddStaticObject(o)
        case "heart"
```

```

        o = CreateItem(HEART_ID, vHeartImage, f.x, f.floorY, f.z)
        EA_AddStaticObject(o)
    case "chest"
        o = CreateItem(CHEST_ID, vChestImage, f.x, f.floorY, f.z)
        EA_AddStaticObject(o)
    ...

```

We will soon change the code where the player picks up items so that it uses the id of the objects instead of the sprite image, but first we should clean up the creation of the player.

The setup of the player is somewhat messy in the previous examples. In example 5 we added a function to create and return an enemy object. I named the function `CreatePumpkinEnemy`. Let's do the same thing with the player object, put all the initialization code in one function. Start by defining a visible variable, `vPlayer`, where you have your other visible statements:

```
visible vPlayer
```

And when when looping through the flags from `EA_LoadMap`, call a new function `CreatePlayer`, that will return a player object:

```

vPlayer = unset
foreach f in flags
    select f.flag
        case "player"
            vPlayer = CreatePlayer(f.x, f.floorY, f.z)
            EA_AddObject(vPlayer)
            EA_SetCamera(vPlayer)
        ...
    endsel
next
assert typeof(vPlayer), "No player flag found"

```

Copy all the code that you used for setting up the player object to that function. My `CreatePlayer` function now looks like this:

```

function CreatePlayer(x, y, z)
    player = EA_FpsPlayer()
    player.id = PLAYER_ID
    player.SetPos(x, y, z)
    player.SetLeap(0.25)
    player.SetHeight(0.6)
    player.SetEye(0.5)
    player.SetMoveSpeed(1.5)
    player.SetMouseSens(0.5)

    player.Update = function(dt)
        if keydown(KEY_F, true)
            res = .Facing()
            if res and res.type = EA_DOOR and res.dist < 1
                if res.data.Open()
                    vOpenDoors[sizeof(vOpenDoors)] = res.data
                    res.data.timer = 3
                endif
            endif
        endif
    endfunction
endfunction

```

```

endif

objs = .SectorObjects()
if objs and sizeof(objs)
    for i = 0 to sizeof(objs) - 1
        if .CollidesWith(objs[i])
            select objs[i].Sprite()
                case vRubyImage
                    vRubies = vRubies + 1
                    EA_RemoveObject(objs[i])
                case vDiamondImage
                    vDiamonds = vDiamonds + 1
                    EA_RemoveObject(objs[i])
                case vHeartImage
                    vHearts = vHearts + 1
                    EA_RemoveObject(objs[i])
                case vChestImage
                    vRubies = vRubies + 10
                    vDiamonds = vDiamonds + 5
                    vHearts = vHearts + 1
                    EA_RemoveObject(objs[i])
            endsel
        endif
    next
endif
endfunc

return player
endfunc

```

Much better! And while we're here, let's adjust the select statement where the player picks up items with new id system:

```

select objs[i].id
    case RUBY_ID
        vRubies = vRubies + 1
        EA_RemoveObject(objs[i])
    case DIAMOND_ID
        vDiamonds = vDiamonds + 1
        EA_RemoveObject(objs[i])
    case HEART_ID
        vHearts = vHearts + 1
        EA_RemoveObject(objs[i])
    case CHEST_ID
        vRubies = vRubies + 10
        vDiamonds = vDiamonds + 5
        vHearts = vHearts + 1
        EA_RemoveObject(objs[i])
endsel

```

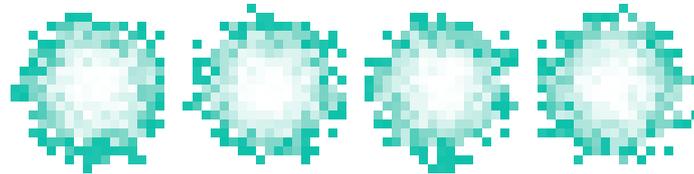
In example_6.n7 I also put some settings, such as field of view and mouse sensitivity in visible variables at the top of the program so that I can change them without searching through the code. That also helps if you chose to implement a settings dialog later.

Let the player shoot

Maybe you have already figured out how to make the player able to shoot? No new engine functions are required for this. But, of course, I'll explain it anyway.

Start by adding a visible variable and load an image for the bullet into it. I'm using an image with four cels, because I want my bullets to be animated.

```
visible vPlayerBulletImage
...
vPlayerBulletImage = loadimage("assets/player_bullet.png", 4, 1)
```



How does a player shoot? Well, he or she clicks the mouse. Should the player be able to shoot as fast as it can click? NO! Slow clickers should not have a disadvantage! Max two clicks per second, I say. Add a field to the player object in the CreatePlayer function, perhaps after you set the id field.

```
player.stimer = 0
```

Then add some logic for the timer and a mouse button check at the bottom of your player's Update function:

```
player.Update = function(dt)
...
.stimer = max(.stimer - dt, 0)
if mousebutton(0, true) and .stimer = 0
    .stimer = 0.5
endif
endfunc
```

Nothing strange there, right? The delta time value (dt) is subtracted from the timer every frame but we use the max function to make sure it stops at 0. And if the player clicks the left mouse button and the timer is 0 we set the timer to 0.5 seconds. We will soon create a function, CreatePlayerBullet(id, image_id, x, y, z, dx, dy, dz). Its first parameter is the id of the object that will be created (maybe we want different types of bullets with different strength later on). The second parameter is the image that the bullet sprite should use. Then comes the position, which should be right in front of the player, and the movement vector, which should be the direction in which the player is looking multiplied by the bullet speed. So:

```
player.Update = function(dt)
...
.stimer = max(.stimer - dt, 0)
if mousebutton(0, true) and .stimer = 0
    .stimer = 0.5
    EA_AddObject(CreatePlayerBullet(
        PLAYER_BULLET_ID,
        vPlayerBulletImage,
        .X() + .DX()*0.2, .Y() - 0.3 + .DY()*0.2, .Z() + .DZ()*0.2,
        .DX()*4, .DY()*4, .DZ()*4))
endif
```

```
endfunc
```

That probably looks worse than it is. We get the player's position with the object functions X, Y and Z. But Y returns the y coordinate of the player's feet. Since the player is not kicking soccer balls at the enemies (... actually kicking soccer balls at pumpkins would have been quite fun), we need to modify the y coordinate a bit. The y axis points downwards, so to make the bullet spawn higher, we have to subtract something – 0.3 works fine. But we also want the bullet to appear slightly in front of the player. The object functions DX, DY and DZ, as explained earlier, returns the x, y and z components of the direction that the object is facing (based on the yaw and pitch angles). So to put the bullet, let's say, 0.2 units in front of the player, we simply add $DX()*0.2$, $DY()*0.2$ and $DZ()*0.2$ to the coordinates. The last three parameters is the movement vector of the bullet. It should travel in the direction that the player is facing at a speed of ... 4 units per second.

Not too hard, right? Good, then it's time for the actual implementation of CreatePlayerBullet!

```
function CreatePlayerBullet(id, img, x, y, z, dx, dy, dz)
    b = EA_Object()
    b.id = id
    b.SetPos(x, y, z)
    b.SetSprite(img, 0, false)
    b.SetHeight(height(img)/64)
    b.SetRadius(0.5*width(img)/64)
    b.dx = dx
    b.dy = dy
    b.dz = dz
```

You know all about setting an objects position, sprite and size. But note that I now set the last parameter of SetSprite(image_id, cel, onlyYaw) to false instead of true. Because when it comes to the bullets, we want them to fully face the camera (the sprites should be rotated both around the y and x axis). You can try setting the parameter to true (as we do for the enemies and items) just to see the difference. Instead of messing with SetYaw and SetPitch and use the DX, DY and DZ functions later on, we create three fields for the direction of the bullet – dx, dy and dz. Everything fine so far? Good, let's go on with two more fields:

```
b.numCels = cels(img)
b.cel = 0
```

As I wrote earlier, I want my bullets to be animated. But the code will work for non-animated bullets too. I add one field for the number of cels in the image (a bit faster than using the cels function later on) and one for the current cel. This is all the data that the bullet object needs. Now add the bullet's Update function, where all the fun stuff happens. I'm pasting it all at once:

```
b.Update = function(dt)
    if .numCels > 1
        .cel = (.cel + dt*30)%numCels
        .SetCel(int(.cel))
    endif

    res = .Move(.dx*dt, .dy*dt, .dz*dt, 0)
    if res.any
        EA_RemoveObject(this)
    else
```

```

objs = .SectorObjects()
if objs and sizeof(objs)
  for i = 0 to sizeof(objs) - 1
    if objs[i].id = PUMPKIN_ID
      if .CollidesWith(objs[i])
        objs[i].Hit(.dx, .dz)
        EA_RemoveObject(this)
        break
      endif
    endif
  next
endif
endif
endif
endfunc

```

The first thing we do is to check if the number of cels, the numCels field, is greater than 1. In that case we update the animation at 30 frames per second and set the sprite's image cel with the object function SetCel, that you've seen before.

Next we call Move using the dx, dy and dz fields and set the leap parameter to 0. If the bullet hit ANYTHING it should be removed. Earlier we have only checked the w field of the object returned by Move. w stands for wall. There is also a g field for ground and c field for ceiling. But we can check a field named any if we just want to know if there was any collision at all. If there was a collision we call EA_RemoveObject. If not, we check for collision with the enemies.

As you know from an earlier example, the object function SectorObjects returns all the objects that are in the same sector as the current object. If the returned value, objs, has a type and its size is greater than 0, we loop through the objects. And if the id of an object is the id of our enemy type, PUMPKIN_ID in my case, we check if the objects intersect. In the case of intersection, call a function, Hit, of our enemy that we have yet not implemented, remove the bullet and break out of the loop.

After the implementation of the Update function, your CreatePlayerBullet function must return the bullet object, b. Here's the entire function:

```

function CreatePlayerBullet(id, img, x, y, z, dx, dy, dz)
  b = EA_Object()
  b.id = id
  b.SetPos(x, y, z)
  b.SetSprite(img, 0, false)
  b.SetHeight(height(img)/64)
  b.SetRadius(0.5*width(img)/64)
  b.dx = dx
  b.dy = dy
  b.dz = dz
  b.numCels = cels(img)
  b.cel = 0
  b.Update = function(dt)
    if .numCels > 1
      .cel = (.cel + dt*30)%numCels
      .SetCel(int(.cel))
    endif
    res = .Move(.dx*dt, .dy*dt, .dz*dt, 0)
    if res.any
      EA_RemoveObject(this)
    else
      objs = .SectorObjects()
      if objs and sizeof(objs)
        for i = 0 to sizeof(objs) - 1

```

```

        if objs[i].id = PUMPKIN_ID
            if .CollidesWith(objs[i])
                objs[i].Hit(.dx, .dz)
                EA_RemoveObject(this)
                break
            endif
        endif
    next
endif
endif
endifunc

return b
endifunc

```

Boop the pumpkin

Only one thing is left now, and that is to make something happen with the enemy when it's hit by a player bullet. As you can see in the code above, I have already assumed the existence of a function named Hit in the enemy object. It takes two parameters, which is the movement vector in x and z of the object that caused the hit. You see, I want the enemy to be bounced away by the bullets. I also want the enemy to flash in white, so that the player gets some feedback and knows that a bullet actually hit something. So I add a new row of cels to my enemy image:



It's a bit hard to tell here, but on the second row the inside of the enemy outline is white. I change the loadimage call to inform the loader that the image has four columns and two rows:

```
vPumpkinImage = loadimage("assets/pumpkin.png", 4, 2)
```

How do we make the enemy bounce away when hit by a bullet? There are many ways to do that. I choose to add a push vector and a parameter. In each frame, the parameter will decrease towards 0. The movement of the enemy will always be affected by this vector multiplied by the parameter. When the parameter is at its max, 1, the push effect is strong, and when the parameter is 0 there is no push effect at all. Here I add the fields for pushing to the enemy object in CreatePumpkinEnemy:

```

function CreatePumpkinEnemy(x, y, z)
    enemy = EA_Object()
    ...
    enemy.push = 0
    enemy.pushDX = 0
    enemy.pushDZ = 0

```

push is the parameter and pushDX and pushDZ the vector. I also add a timer field for the flash effect:

```
enemy.htimer = 0
```

Let's jump to the implementation of the Hit function, called by the player bullet objects. Add this function to the enemy object after the update function (or before, doesn't matter):

```
enemy.Hit = function(dx, dz)
    .pushDX = .push*.pushDX + dx
    .pushDZ = .push*.pushDZ + dz
    .push = 1
    .htimer = 0.15
endfunc
```

A bit more code than expected? If there is already a push effect going on, we add "what is left" of that effect to the new push vector by multiplying the current vector with the push parameter. Then we set the parameter to 1 and the flash timer (htimer) to 0.15 seconds.

Now, start by adding code to handle the flash effect to the enemy's Update function:

```
enemy.Update = function(dt)
    .cel = (.cel + dt*8)%4
    if .htimer > 0
        .SetCel(int(.cel) + 4)
    else
        .SetCel(int(.cel))
    endif
endfunc
```

The first line, the regular animation of the enemy, is unchanged. In the if statement that follows, add 4 to the cel value to use the flash version of the animation if the htimer field is greater than 0.

Then we need to modify the call to Move in the same function, from:

```
res = .Move(.DX()*0.75*dt, 1*dt, .DZ()*0.75*dt, 0.25)
```

, to:

```
res = .Move((.DX()*0.75 + .push*.pushDX)*dt, 1*dt, (.DZ()*0.75 +
.push*.pushDZ)*dt, 0.25)
```

As you can tell, if the push field is 0, there will be no effect.

The last thing we need to do, at the bottom of the enemy's Update function, is to decrease the push and htimer fields towards 0:

```
.push = max(.push - 1.5*dt, 0)
.htimer = max(.htimer - dt, 0)
```

endfunc

I decrease the push parameter by 1.5 per second, so that the effect lasts for less than a second.

Even more ...

I'm sorry, that was a lot of code and barely any pictures. You probably don't need to understand all of it. Was the push effect overkill? Wouldn't it have been enough with the flash? Perhaps, but it looks and "feels" cool. If you want to, you could probably add stamina to the enemies on your own and make them go bye bye when their stamina reaches 0, am I right? Init a stamina field to, let's say, 3 and decrease it in the enemy's Hit function. When the stamina reaches 0, call EA_RemoveObject and the enemy is gone. On the other hand, an enemy couldn't possibly disappear without a cool animation! Could you pull that off?

In example_6.n7 I have added even more stuff than I've shown here. The player holds a wand that bobs as the player walks. I've added sound effects and some music that will make your brains collapse. So I suggest you have a good look at the code :)