# PlaneSmith - Quick Guide

# Overview

PlaneSmith is a level editor for 2D games. It is general purpose, meaning it doesn't make any assumptions about the games you make (other than them being 2D).

One of the thing it offers is the ability to design the format of the output you get. Do you want an XML file? A piece of C++ code? Or maybe a simple list of integers and strings that your program will interpret? PlaneSmith lets you do all that!

How does PlaneSmith work? A PlaneSmith project contains a dictionary and levels. A dictionary has a set of definitions. A level has a set of objects. Each object is associated with one definition. You can think of definitions as types and objects as instances of those types. Multiple levels can use the same dictionary.

Every level, definition and object is assigned a piece of code. This code is what specifies the final output.

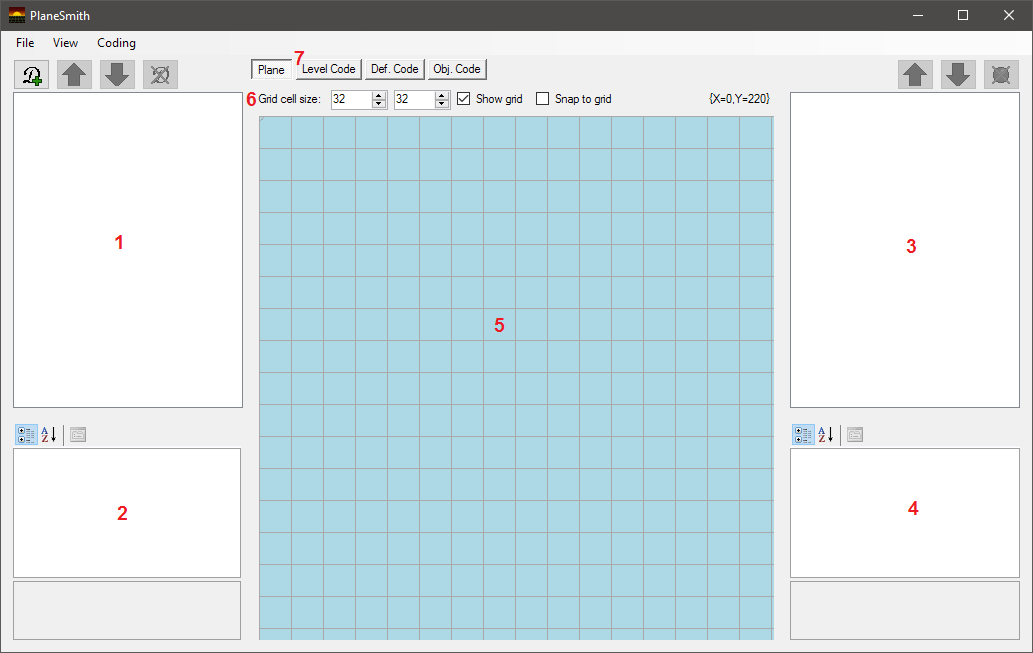
This document will guide you through a creation of a simple project in PlaneSmith. You can open the finished projects from the example folder. There are two examples. Their levels are the same, but one generates C++ and the other XML output.

In the same folder as this document, you will find a text detailing how to code. You should read it after you've gone through this guide.

# Startup

*Since PlaneSmith levels remember the path to their dictionary as relative paths, it's a good idea to keep all level and dictionary files of a project under a single folder.*

Open PlaneSmith. You will see a screen like this.



**1** List of all current definitions. Above it are buttons for manipulating this list.

**2** Properties of currently selected definition.

**3** List of all objects in the level. Above it are buttons for manipulating this list.

**4** Properties of currently selected object.

**5** This is the plane. It is a visual representation of the level.

**6** Options regarding plane grid (gray lines). Grid is visual only.

**7** Tabs other than Plane contain textboxes for coding.

First, we will design a level and then jump into some coding.

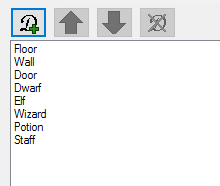
*If you want to open a project, click on File/Open menu item. Loading a level file will also load the dictionary, whereas loading a dictionary will not load any levels.*

# Example project - Level design

Click on the **Add Definition** button. Input the definition name and choose an image file. *When you create a definition it is automatically selected. You will see an object following your cursor on the plane. Right-clicking on the plane will deselect definitions.*



In this example we will have the following definitions. .png image files are located under **Examples/Sprites** and have corresponding names.



*To delete a definition, select it from the list and click on Delete Definition (right-most button).*

Now let's design our level! We will use a 32x32 grid. Check **Snap to grid**. This way objects will be aligned to grid lines.

Untitled

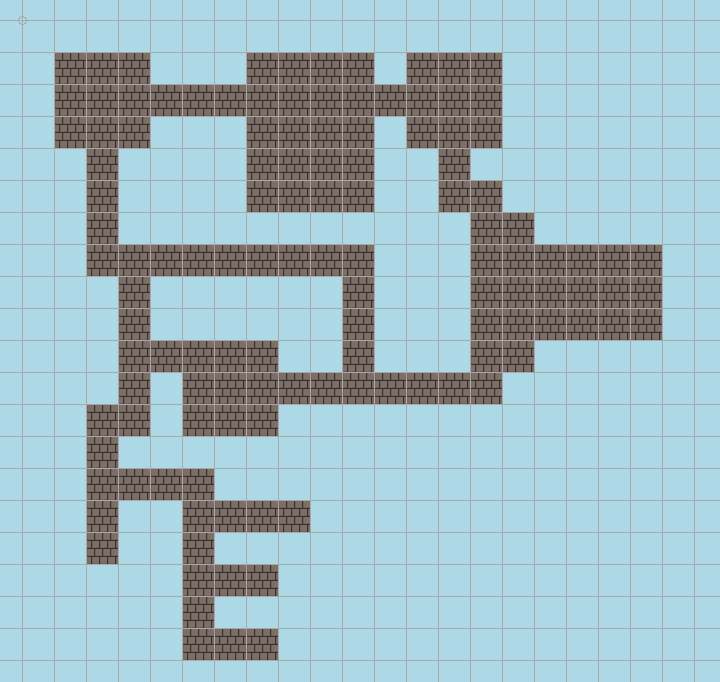
Select **Floor** definition from the list.

*Objects are placed by left-clicking inside the plane. You can hold shift and left mouse button to drag-place multiple objects, though only when Snap to grid is active.*

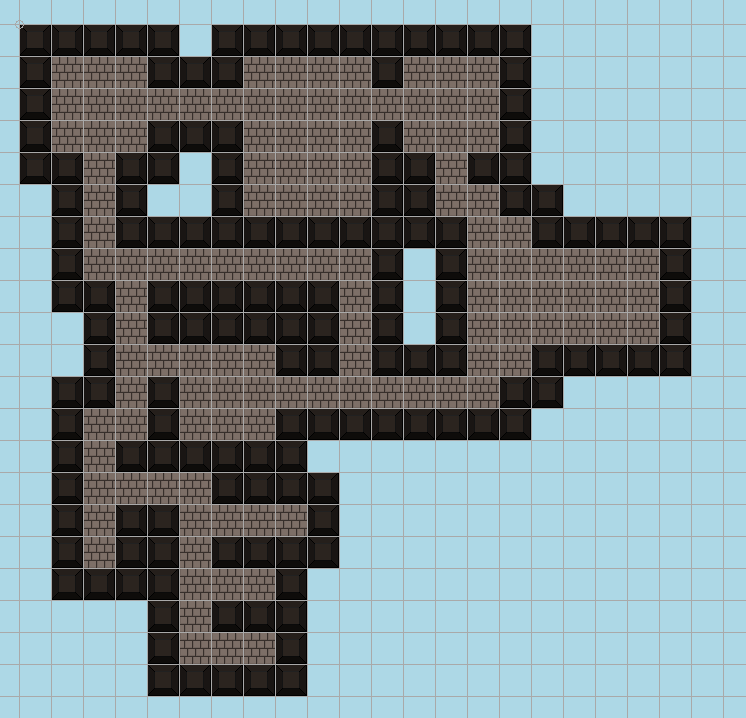
*Right-clicking deselects a definition. Right-clicking again deletes an object at cursor (top-most object). You can left-click an object to select it, further left-clicks will cycle through objects found at that location (if you have multiple objects on top of each other).*

*By holding the middle mouse button (i.e. wheel) and dragging around the plane, you can move your view around. Double clicking the plane with the middle mouse button will reset any such movement.*

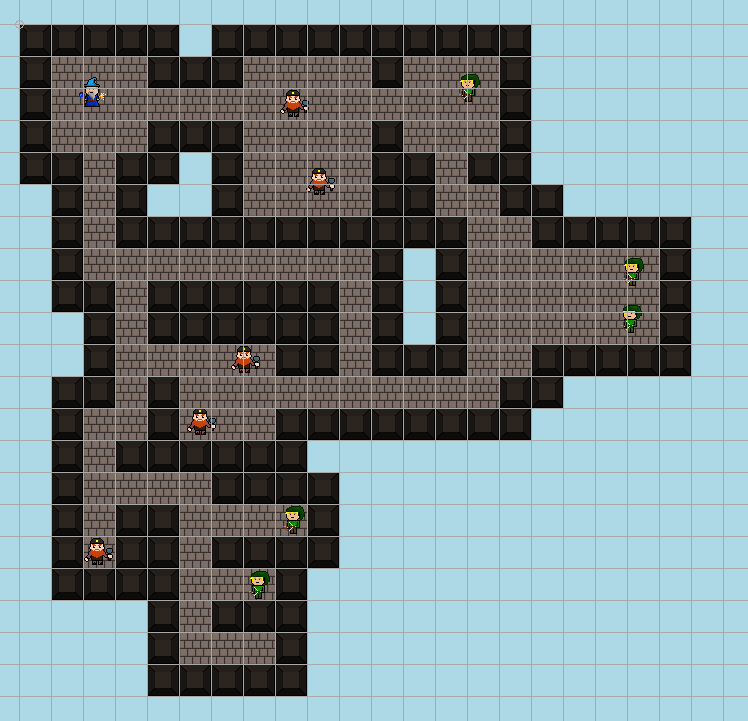
Fill out the level with some **Floor** objects.



Now, let's surround them with walls. Select the **Wall** definition and place them on the plane.



Now uncheck **Snap to grid** and add a **Wizard** (the player), some **Elves** and **Dwarfs**. In our imaginary game, characters won't be bound to tiles.



For fun, let's add some loot. Check **Snap to grid** again and add **Potions** and a **Staff**.

While you're at it, add some **Doors** too. If you change your mind about the layout, you can poke a hole in a wall by right-clicking on it to delete.



Ok, this is starting to look like an actual level!

At this point, it would be a good idea to save your project (with **Ctrl+S**). *First you will save the dictionary and then the level. Preferably, group all levels and their dictionary inside the same directory (because of level's relative pathing to its dictionary).*

# Example project - Coding

Let's say we are making an RPG in C++ and need to implement the **setupLevel** method of our class **Level1**.

Go to **Level Code** tab. Level's code is the starting point from which output is generated. Here we define the outline of our sorry code. Copy this into the textbox:

#include "Level1.h"

«FECH.DEF.

»#include "«NAME».h"«END»

void Level1::setupLevel() {

«FECH.DEF.

»«CODE»«END»

}

What's going on here? Aside from the obvious C++, PlaneSmith statements are written inside **«** and **»**. *Click* ***Ctrl+B*** *to quickly add them to your code.* Everything outside of these is considered a terminal string and will be used as-is.

The first statement is **FECH**. Name **FECH** is a mix between fetch and for each. Since we chose to fech through definitions (with **DEF**) it will cycle through all known definitions and output something related to them. In this case it is printing include directives for our C++ code.

Notice the newline after the second **.** in **FECH**. This is a split terminal, which is inserted between every two definitions. In this case we want to separate includes in different lines of code.

A **FECH** statement is closed with an **END** statement. Everything inside those two is the body of the **FECH** statement.

**«NAME»** is an attribute statement. Here, the program will generate the name of the definition, for every definition it cycles through.

Further down, we have another **FECH**. This one prints the code assigned to our definitions and spreads them with two newlines.

To specify definition codes go to **Def. Code** tab. You will see two textboxes. The upper one is for definition codes. All definitions in this project will have the same code:

«FECH.OBJ.

»«CODE»«END»

Here we cycle through all objects belonging to this definition and print their codes. They are separated by a newline.

The lower textbox has a code that will automatically get assigned to new objects belonging to this definition. This way you don't have to copy the same code into every new object.

This will be the code for **Floor**:

makeFloorAtTile(«X» / 32, «Y» / 32, "assets/sprites/«DEF.NAME».png");

It's a made up method for putting a floor on a tile. The first two PlaneSmith statements return object's coordinates on the plane. The third one will return the name of this object's definition.

Codes for **Wall**, **Door**, **Potion** and **Staff** are similar. You can find them in the example project.

Because we've already placed our floor objects, this code isn't added to them. It is only added to new objects that get placed. What you will do now is copy this code into every floor instance in our level.

Nah, just kidding. Click on the **Reinsert obj. codes** button below to let the program do it for you.

Code for dwarves is:

{

Enemy \*dwarf = new Dwarf();

dwarf->setPosition(«X», «Y»);

addEnemy(dwarf);

}

Curly brackets will avoid conflicts with other dwarf variables. Codes for **Elf** and **Wizard** are similar. Don't forget to reinsert them into existing objects.

If you go to **Obj. Code** and select an object you will see its code. For exercise, let's change codes of two specific objects.

Go to **Plane** tab. Select one of the doors (e.g. the one next to the staff). *You can select it by left-clicking on it. If you selected a definition, first deselect it by right-clicking.* Now go to **Obj. Code** tab. Add this line:

getTile(«X» / 32, «Y» / 32)->getDoor()->setLocked(true);

Now go back to **Plane** and select one of the elves, and change its code to:

{

Enemy \*elf = new Elf();

elf->setPosition(«X», «Y»);

elf->getLoot()->addKey();

addEnemy(elf);

}

As you see, PlaneSmith allows you to alter behaviours of individual objects in your level.

Finally, generate your code (with **Ctrl+Shift+B**). Your project will first be saved. If you made no errors you should get a file with a C++ method implementation.

You can compare it to the output from the example project. Consider that the same output could have been generated with a shorter code. There is another example with the same level, but which generates an XML file.

# Conclusion

And there you have it - the basics of how to use PlaneSmith. You should know read the documentation on coding.

There will probably be bugs, the UI could use some tidying up and some new features would be welcome. If you have any suggestions or feedback let me know. This project is open-source on GitHub: <https://github.com/OnionBurger/PlaneSmith>.

Thanks for reading and enjoy!