# Recap

In the last article we enabled the multi-level game play and the game is almost finished – all it needs is some polish to make it look a little professional. In this article, we will add some animation to the splash screen and some sound effects before packaging the application ready for publication.

Callout to “Const vs #defines vs enumerations”

# Splash Screen Animation

When I originally designed the artwork for the splash screen, I made individual sprites for each of the pipe pieces rather than one large graphic allowing me to ‘animate’ the pipes being laid.

In the previous article, the drawSplash() routine clears the screen and renders all of the pipes in one action.

arduboy.clear();

sprites.drawOverwrite(112, 0, logo\_straight\_TB\_noflange, frame);

sprites.drawOverwrite(112, 16, logo\_elbow\_TL, frame);

sprites.drawOverwrite(96,  16, logo\_elbow\_TR, frame);

sprites.drawOverwrite(96,  0, logo\_elbow\_LB, frame);

...

arduboy.display();

The code below shows the modified version of the drawSplash() routine. It is similar to that original except it uses a helper function, splashAnimation(), to render the pipes individually. As you will see in a moment, the splashAnimation() routine takes the original parameters as the original sprites.drawOverwrite() function it replaces but adds an additional one that allows the functionality to be skipped. After seeing the animation once or twice, most players will be happy to skip straight to the game play.

bool skipSplash = false;

arduboy.clear();

arduboy.display();

skipSplash = splashAnimation(112, 0, logo\_straight\_TB\_noflange, frame, skipSplash);

skipSplash = splashAnimation(112, 16, logo\_elbow\_TL, frame, skipSplash);

skipSplash = splashAnimation(96,  16, logo\_elbow\_TR, frame, skipSplash);

skipSplash = splashAnimation(96,  0, logo\_elbow\_LB, frame, skipSplash);

...

splashWaitForever();

The splashAnimation() function is shown below. As mentioned, it accepts a new parameter, skip, and if detected to be true causes the function to immediately return. Otherwise, the requested sprite is rendered to the screen and the routine loops 20 times, pausing for 15 milliseconds at the end of each loop. If the player has pressed the ‘A’ button, the function returns true and all subsequent calls to the function from the drawSplash() routine effectively skip the animation.

#define ANIMATION\_DELAY\_SHORT     20

bool splashAnimation(byte x, byte y, const uint8\_t \*bitmap, uint8\_t frame, bool skip) {

  if (skip) return true;

  int i = ANIMATION\_DELAY\_SHORT;

  sprites.drawOverwrite(x, y, bitmap, frame);

  arduboy.display();

  while (i >= 0) {

  arduboy.pollButtons();

    if (arduboy.justPressed(A\_BUTTON)) { return true; }

    arduboy.delayShort(15);

    i--;

  }

  return false;

}

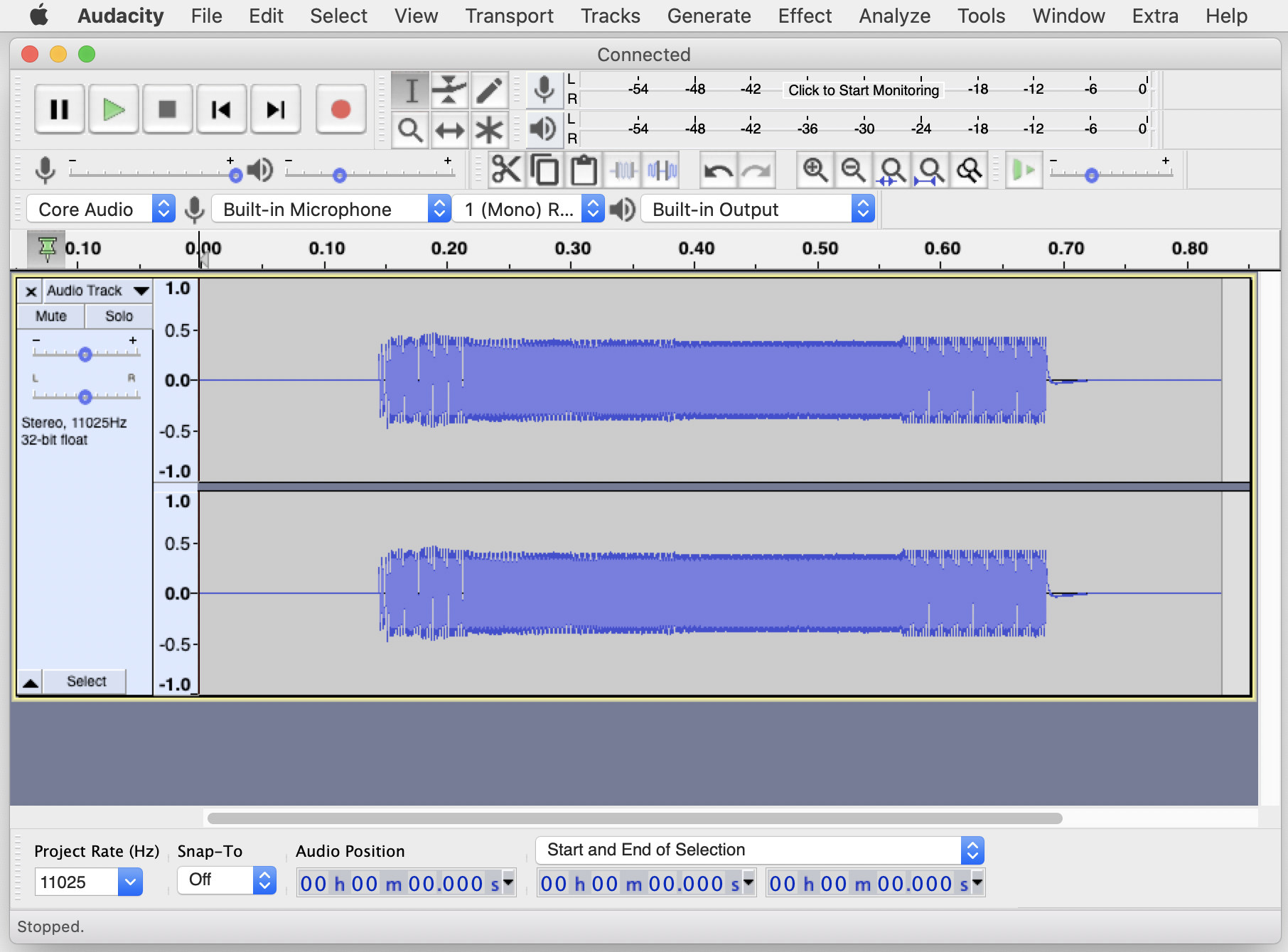
We will revisit these two functions in a moment to add some sound effects and additional functionality that will allow the player to turn the sound effects on or off.

# Adding Sound Effects

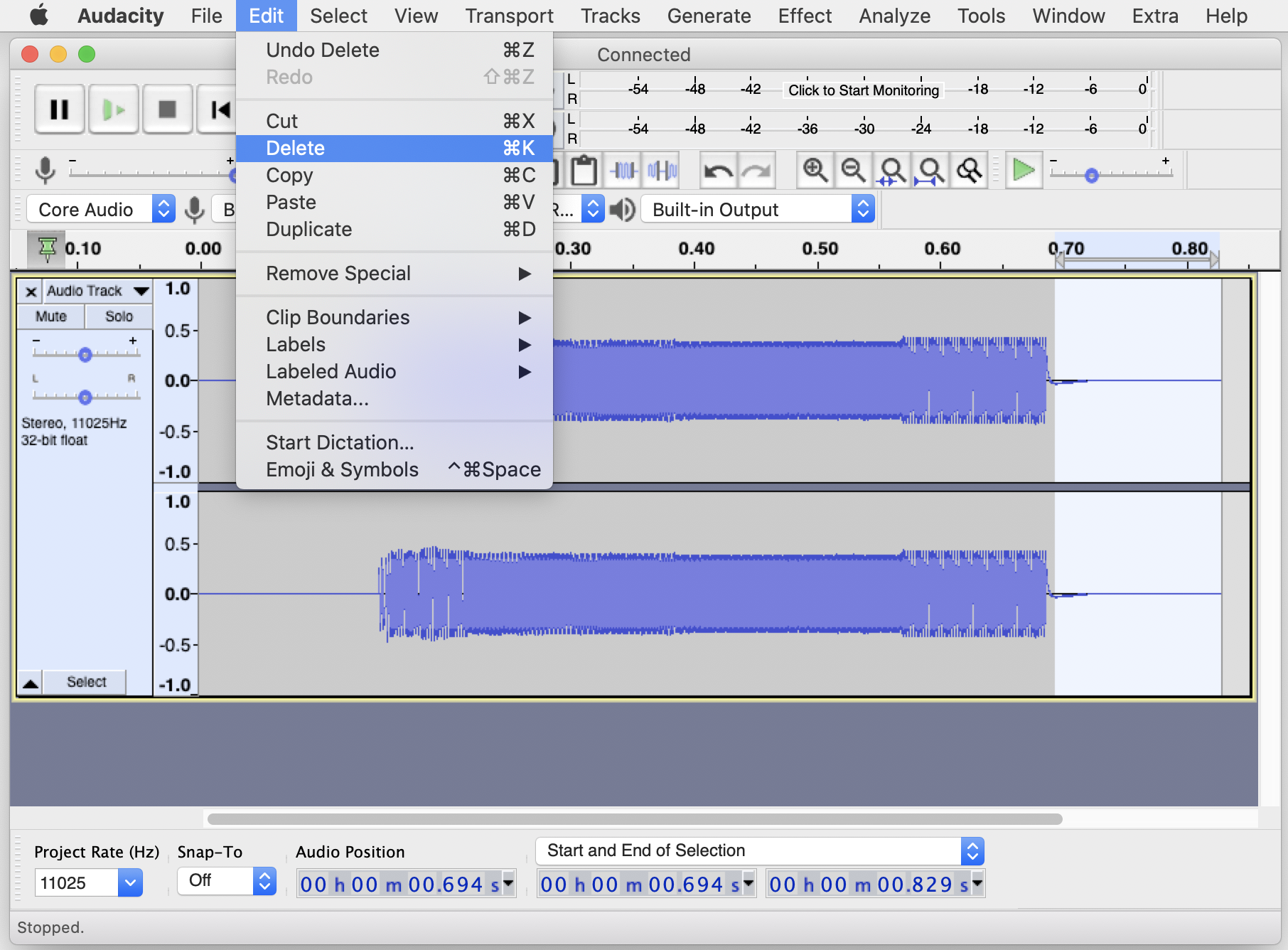
Adding sound effects to any game really livens it up. You can find literally thousands of free sounds online and these are typically provided in WAV or MP3 format. The following instructions detail how to convert the file into a .RAW format for use with the Pokitto.

I have used an open source tool called Audacity which is available for Windows, Macintosh and GNU/Linux and can be downloaded from <https://www.audacityteam.org/download/>

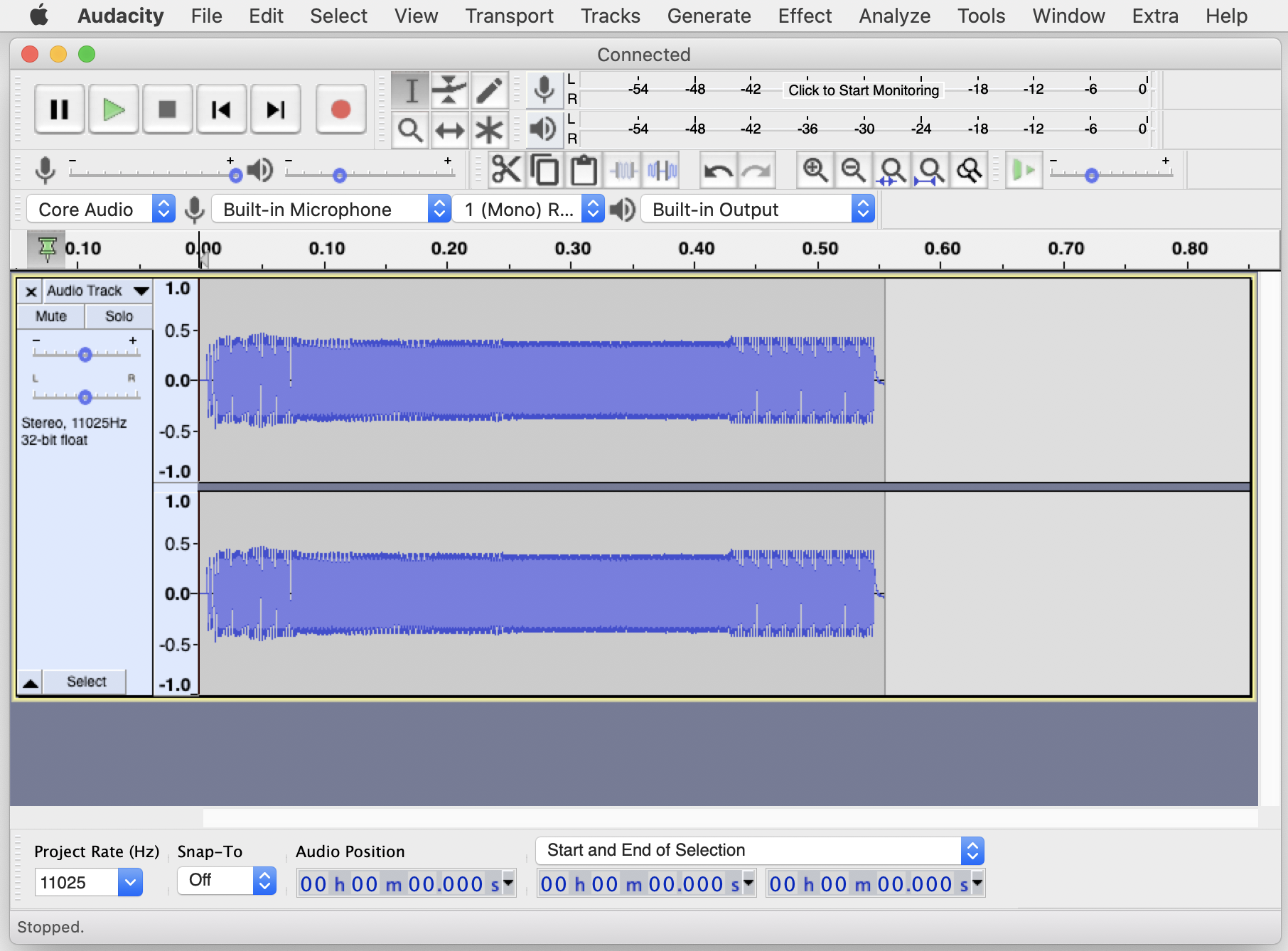
After downloading and installing Audacity, open the sound effect file you wish to convert. As mentioned, sound effects can be found freely on the internet but you should look for simple, short sound effects as the converted sounds can be extremely large and cause memory issues with you program. Depending on the sound you chose, you may see one or two tracks (mono or stereo) as shown below.



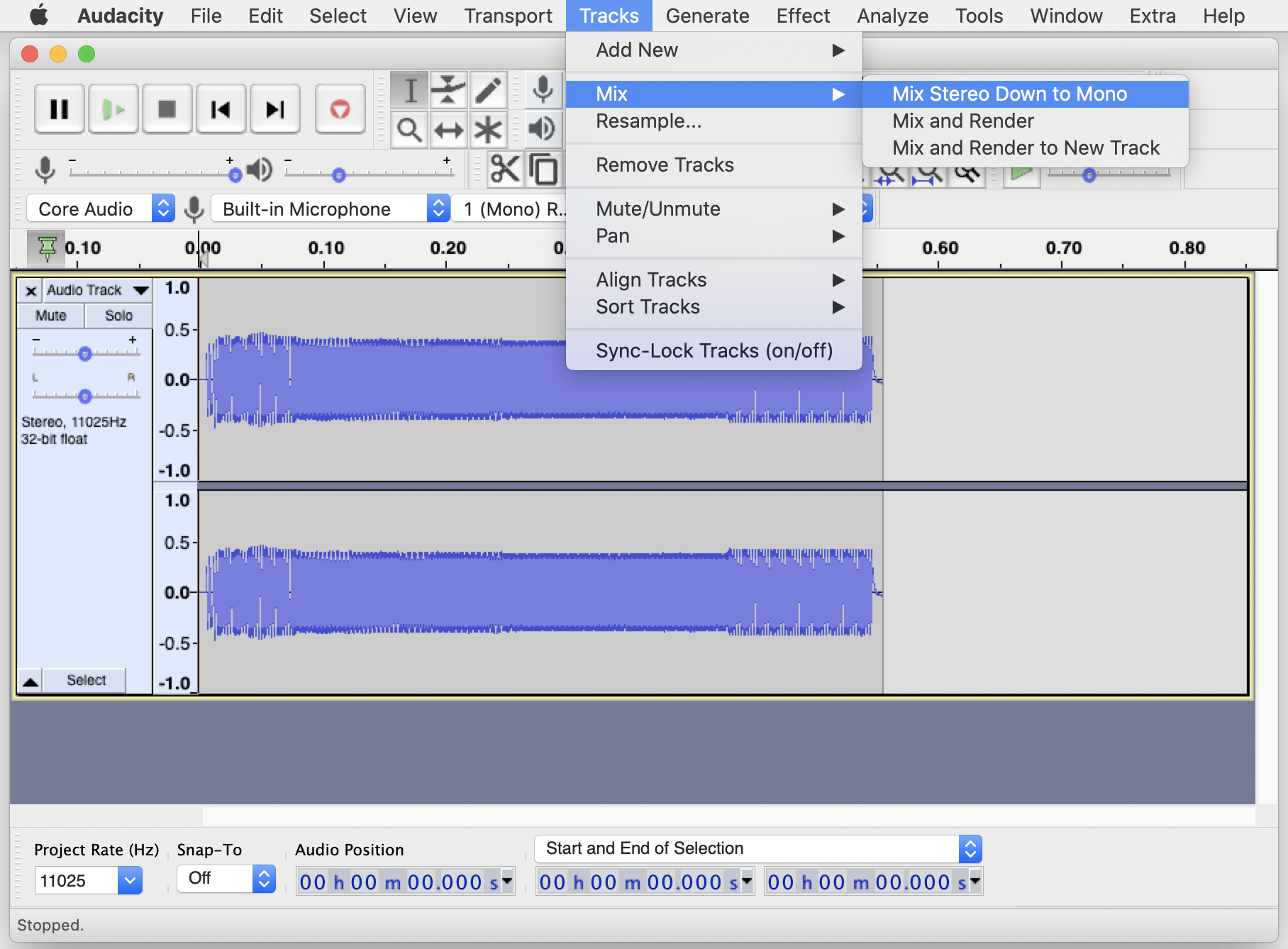
You may notice that there is a second or two of silence at the start and end of the sound effect. The .RAW format does not have any compression so silence takes up just as much memory in a sound effect as the actual sound itself. Highlight the silent areas and delete them via the *Edit /* *Delete* menu option, as shown below.



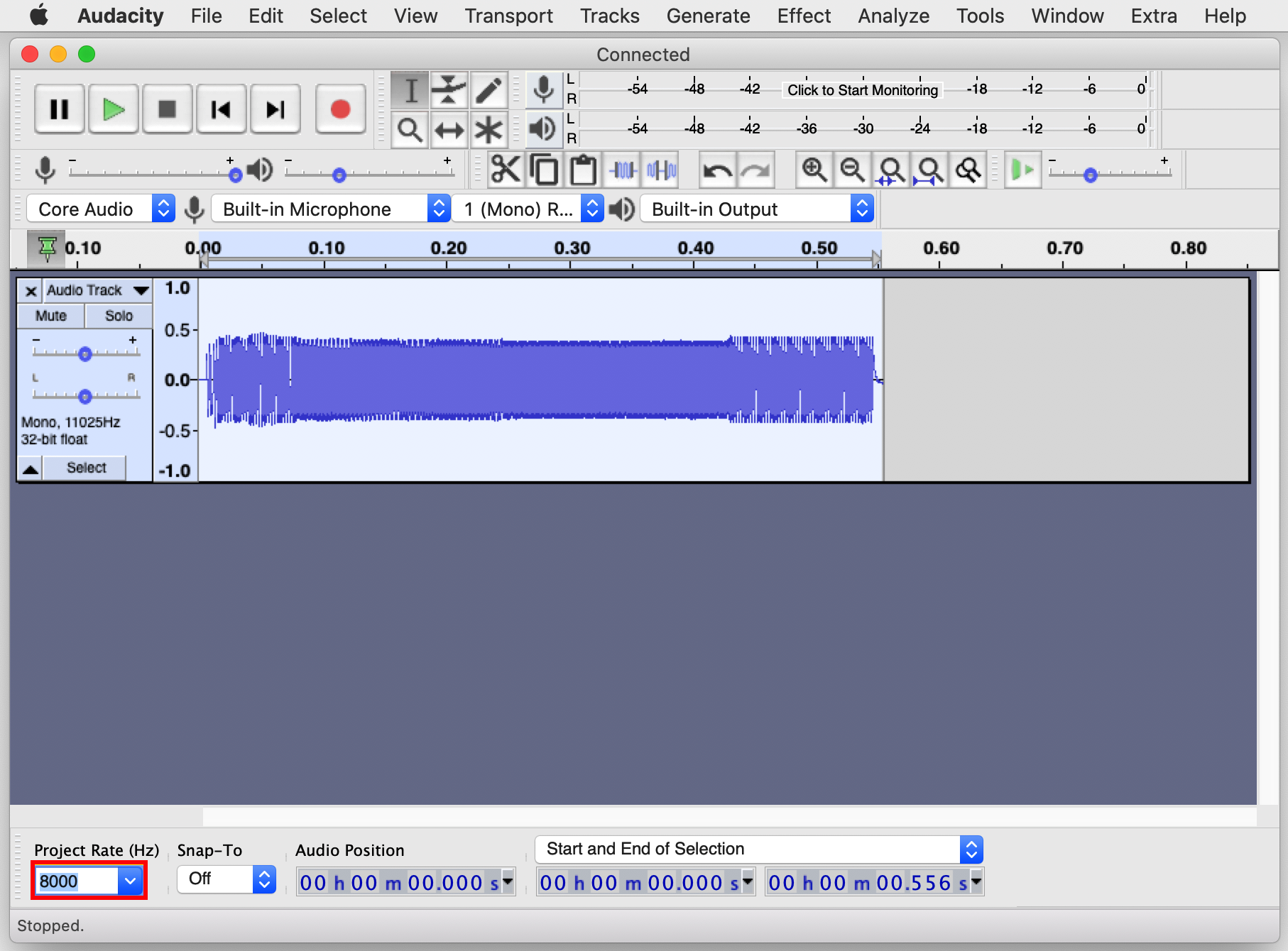
Trim the silence from either end of the sound effect if necessary to shorten the track to a minimum, as shown below.



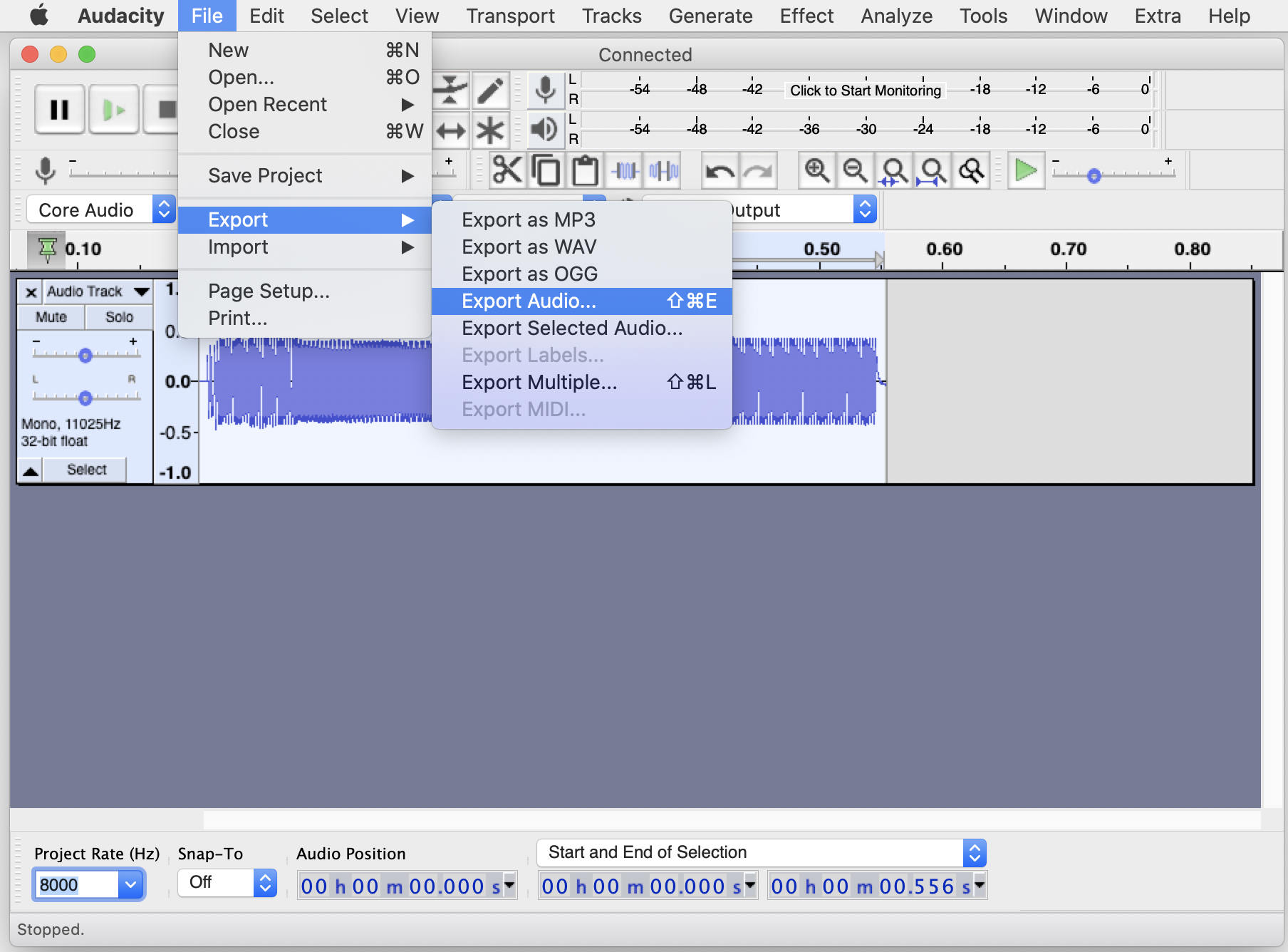
The .RAW format can only support mono sounds so compress the two tracks into a single. Mono track using the *Tracks > Mix > Mix Stereo Down to Mono* menu option.



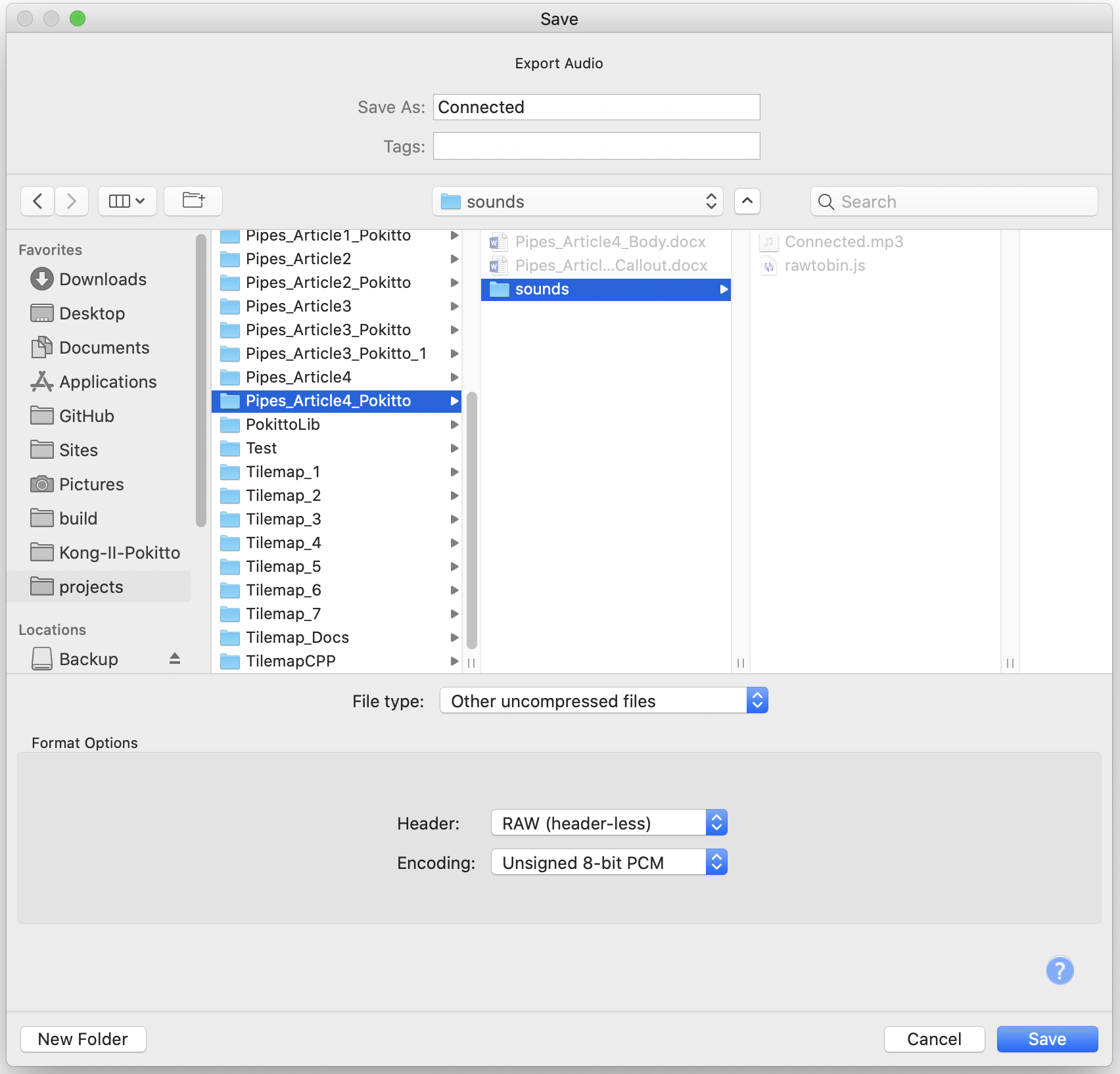
Next, set the *Project Rate* to *8000*. This will produce the smallest file but at the cost of sound quality. You could experiment with different values to see how big the files are and if the difference in sound quality is noticeable.



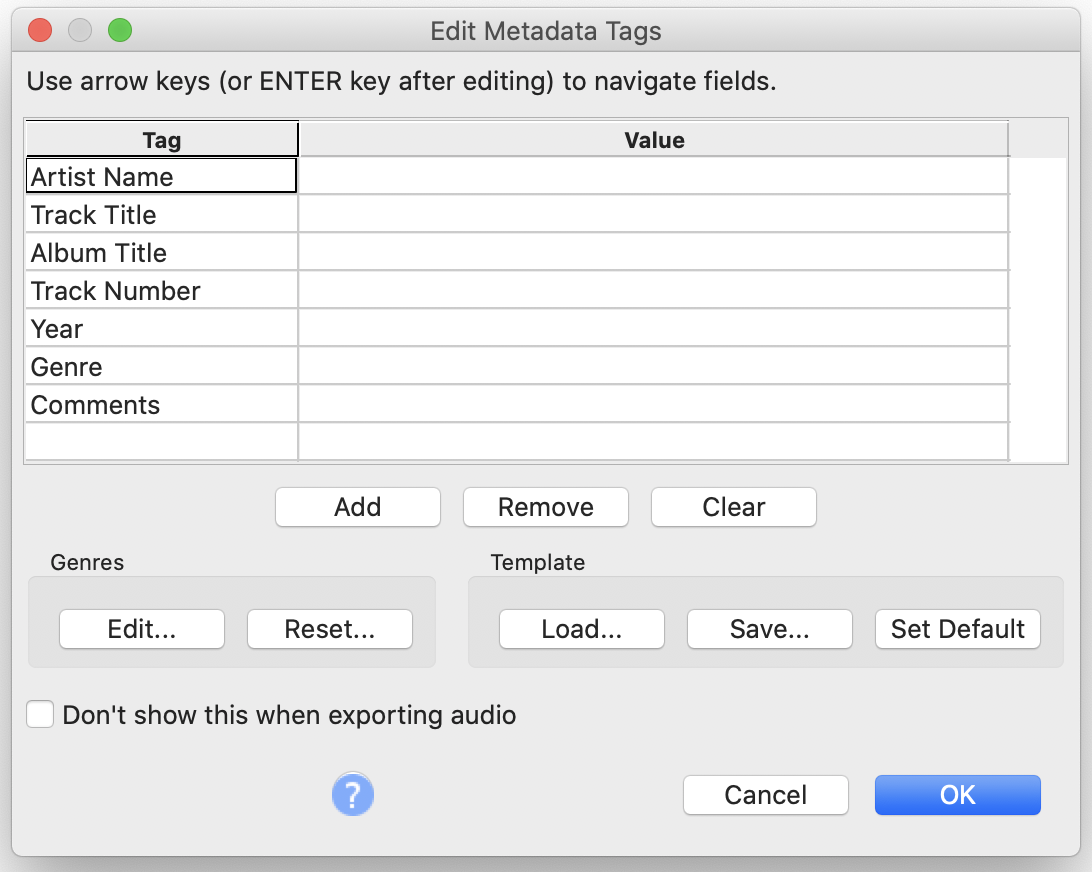
Export he final sound effect by selecting *File > Export > Export Audio.*



Ensure the *Header* option box is set to *RAW* and the *Encoding* option to *Unsigned 8-bit PCM.* Choose a location and filename to store the converted sound effect and then click *Save*.



Leave the *Edit Metadata Tags* table empty and click *OK*  to complete the task.



You will now have a .RAW file exported to disk. To convert it into a source file that the Pokitto can use, we can use a simple utility written by @HomineLudens. It requires a Node.js runtime which can be downloaded from <https://nodejs.org/en/download/>

Ensure the script (rawtobin.js) and the .RAW file you wish to convert are in the same directory. Open a command window and type in the command *node rawtobin.js Connected.raw* (obviously you will need substitute your sound file name in place of *Connected.raw*).



Opening the file in a text editor will show you the results of the conversion – a sample is shown below. Note that the file contains an array of date (the actual music itself) and a second parameter that contains the sound effect’s length. These two variables will be needed when we play the tune later.

const unsigned int sfx\_Connected\_length = 4446;

const unsigned char sfx\_Connected[] = {

89,82,84,87,91,93,97,99,101,104,105,113,159,174,171,169,

164,162,158,156,153,152,149,148,107,81,84,85,89,92,96 ..

In order to play the sounds from our application, we need to add a few settings into the *MySettings.h* file as shown below. The first setting is self-explanatory and instructs the Pokitto to enable sound effects. The second setting tells the Pokitto at what rate (or frequency) the sound files were sampled at. This setting should match the *Project Rate* chosen when exporting the file in Audacity. If you chose a higher sampling rate, such as. 11025Hz, then update this setting to match.

#define PROJ\_ENABLE\_SFX 1

#define PROJ\_AUD\_FREQ 8000

To play a tune, we simply include the converted sound effect file and then call the playSFX() function from the sound library passing the sound effect array and length as parameters.

#include "sounds/Connected.h"

PS::playSFX(sfx\_Connected, sfx\_Connected\_length);

At this point, our application has a funky splash screen, sound effects and an ability for the player to turn the sounds on or off. The complete code is included in my repository at <https://github.com/filmote/Pipes_Article4_Pokitto> and I encourage you to download it and look at some of the other little additions I have made to make the level and puzzle selection and game over banners nicer. This complete version has 30 puzzles per level for you to try out.

# Packaging a game for Distribution

Once your game is complete and testing finished, you will probably want to share the code with the world. There are three ways to do this and I suggest you do them all!

Publishing the code to GitHub

Callout to “What is GitHub?”

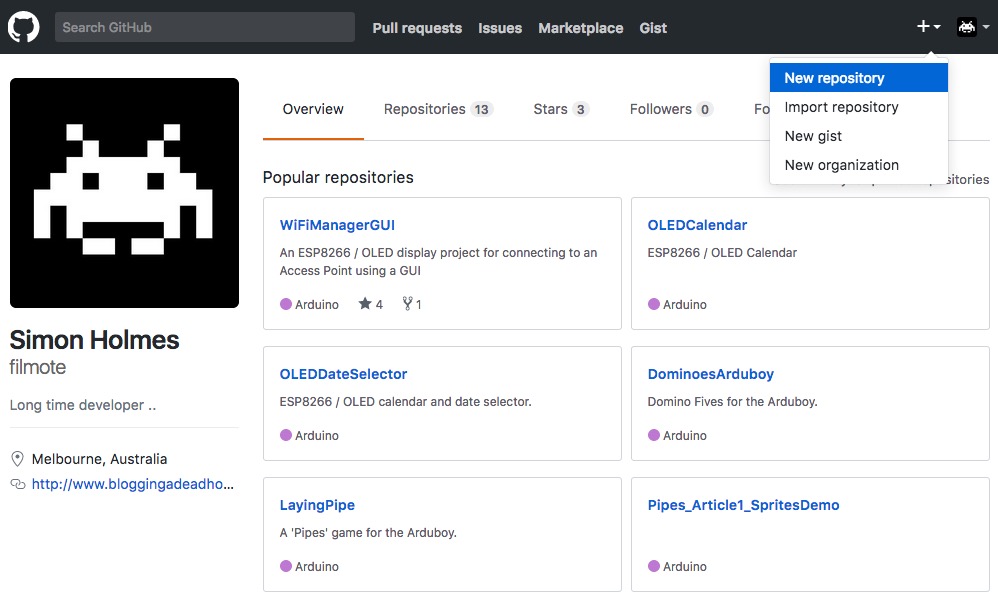
The Pokitto is an open-source project and is supported by a community who will offer assistance and guidance free of charge. As member of the community, you can contribute by publishing your source code for others to learn from.

GitHub is an open-source, web-based version control system that will allow people to view and comment on your code and – if you allow them to – suggest changes that you can incorporate back into the code set. Publishing an application is simple and I have documented it using the web interface only. Once you are familiar with the concepts, I would recommend you download the GitHub Desktop application. Among other things, it compares the files on your desktop against the repository to determine which ones to update thus minimizing the chance that a change will go unpublished or an incomplete change that spans multiple files is published.

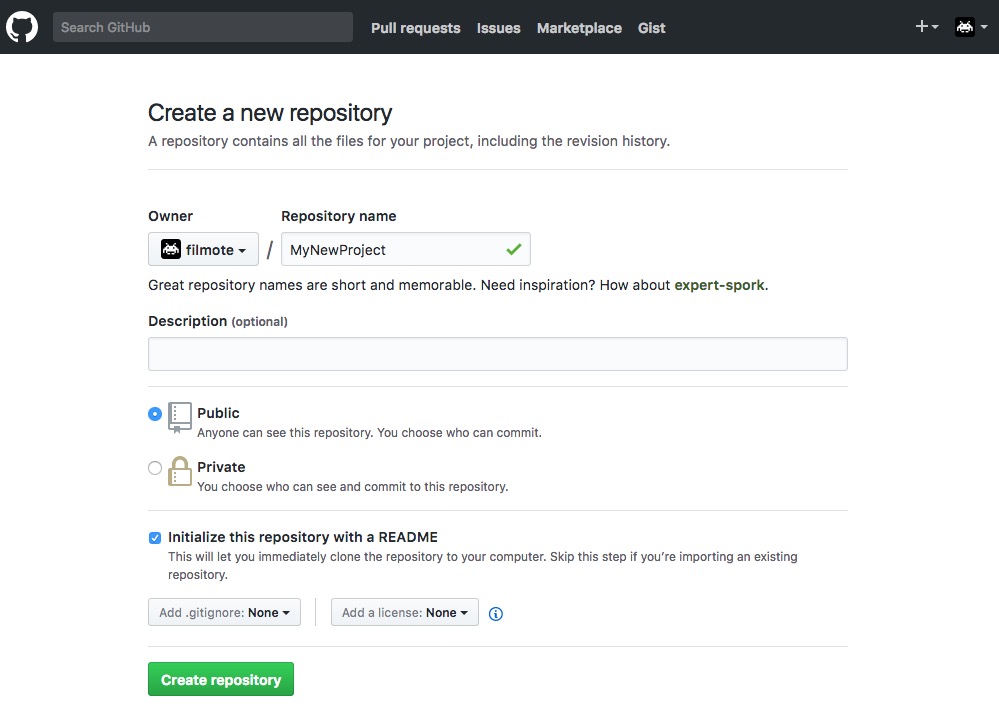
To publish an application:

**Step 1**: Create a user profile if you do not have one. Visit <https://github.com/> to get started.

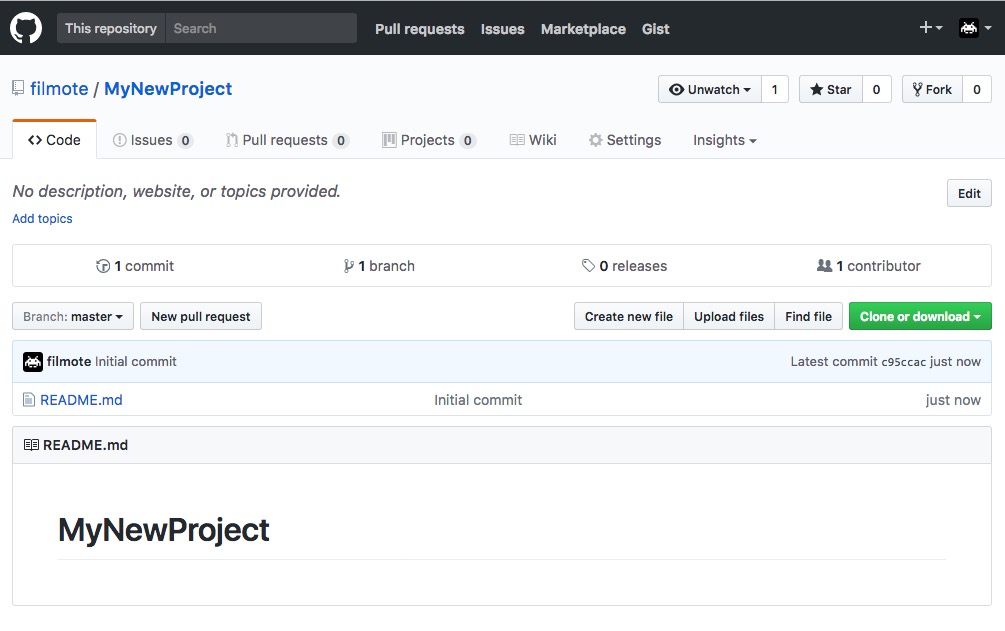
**Step 2**: Log in to GitHub using the credentials created in Step 1. Select *New Repository* from the menu in the top right hand corner of the screen, as shown below.



**Step 3**: Give the repository a name and, optionally, a description. Select the ‘Public’ repository option and check the *Initialize this repository with a README* checkbox. Click the *Create Repository* to complete the creation of the repository.

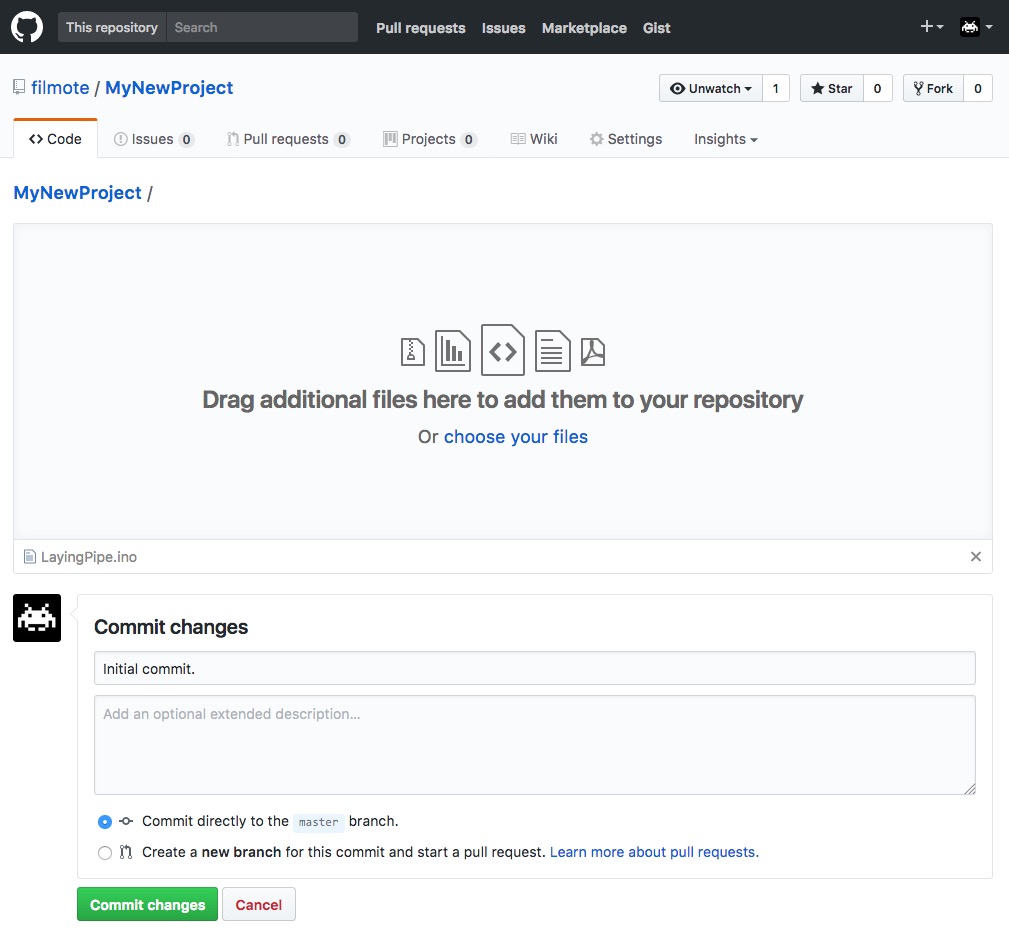


**Step 4**: Give the repository a name and, optionally, a description. Select the ‘Public’ repository option and check the *Initialize this repository with a README* checkbox. Click the *Create Repository* to complete the creation of the repository.

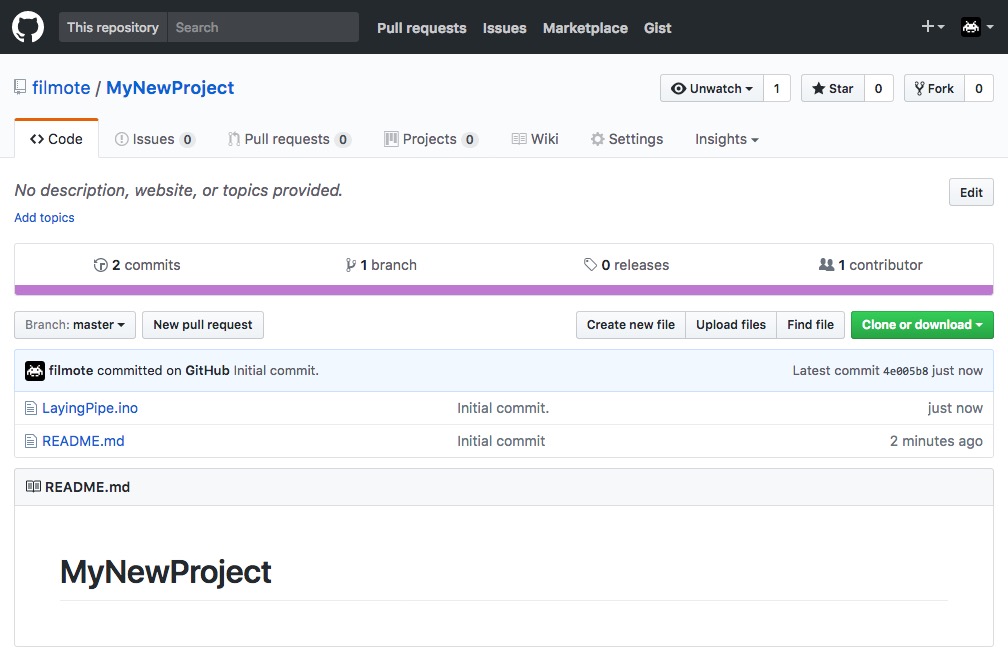


**Step 5**: Once the repository has been created, the actual source files can be added to the project. From the repository view, click *Upload Files* button to begin adding files.

Files can be dragged onto the window from Windows Explorer or OS X’s Finder. The image below shows how the files are accumulated under the drag-and-drop window as they are added. Provide a description of the files being committed and click the *Commit Changes* button.



**Step 6**: The repository is shown again with the files added. You can add new files as needed or update the existing ones using the *Upload Files* button. Other people can retrieve a copy of the code using the *Clone or Download* option.



# Creating a BIN and POP File

A .bin file is a compiled, binary version of your program that can be directly load onto a Pokitto. A compiled binary saves the others the hassle of downloading your code and any prerequisite libraries and having to compile the code and upload it themselves.

A POP file is a zipped file that contains your .bin file, a program description, icons and screenshot images that can be loaded onto the Pokitto’s SD card and displayed using tools such as the Kraken Loader. This format allows the loader to show images of the game in action allowing the user to select and load in a nice rgpahical UI.

Binary files are created automatically as part of the compilation process in FemtoIDE. The compiled .bin file is stored in the root directory of the application and can simply be copied onto a physical Pokitto’ SD card.

## ~~Next Month~~

Actually, there is no next month. I hope you enjoyed this series and learnt something! If you have any troubles understanding a section, please ask a question via the Arduboy Community website.