11
Escaping Singleness – Communicating with Other Applications

In this chapter, we will talk about a couple of scenarios where our Cinder app would be used just as a part of something bigger. This could be a real-time visual solution that requires the power of Cinder and the ease of use of a ready-made program. It could be a collaborative project where one of the involved parties has to do the sound, and another has to handle the visual side with one rule in mind that both sides have to exchange data in real time.

In this chapter, we will create two projects that will show us how to perform the following tasks:

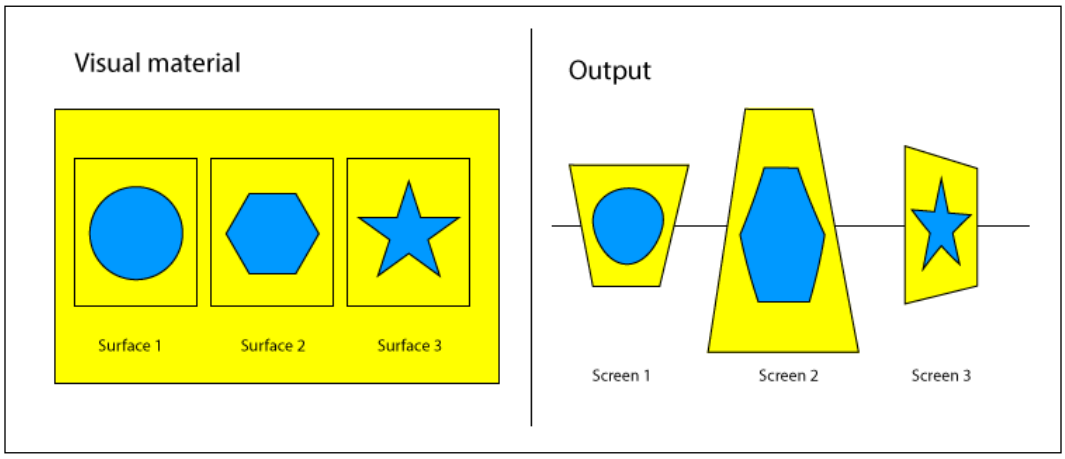
- Display dynamic text by using the Cinder `TextBox` class
- Transfer images from a Cinder application to any other Mac OS X application that supports Syphon (Mac OS X only)
- Learn how to use our application as a source in MadMapper
- Understand the benefits of the **open sound control** (**OSC**) messaging system
- Use pure data to create a simple OSC-based sound application
- Create a Cinder application that can communicate with our pure data application by sending and receiving OSC messages

Using Syphon with Cinder

This section is applicable to the users of Mac OS X 10.6+ (Snow Leopard) only.

Let's say we want to create a visual Cinder application to use in a projection mapping project. **Projection mapping** is a projection technology that turns real-world spatial surfaces into display surfaces. To do that, a specialized computer software is used to spatially map out a two- or three-dimensional object in a virtual environment. The adjusted virtual environment is then projected on the spatial surface by using a projector.

You might use this in a situation where there are not a lot possibilities to adjust the position of a projector, or you have to project visuals on multiple surfaces and you don't want to adjust the material each time the placement of the surfaces changes. You can cut out specific parts of the visual material and adjust their position and size in the output.



We will use a program called MadMapper to spatially map out an image that will be sent from a Cinder app. To do that, we will need to use a software piece called Syphon.

As mentioned on `http://syphon.v002.info/`:

> *Syphon is an open source Mac OS X technology that allows applications to share frames – full frame rate video or stills – with one another in realtime. Now you can leverage the expressive power of a plethora of tools to mix, mash, edit, sample, texture-map, synthesize, and present your imagery using the best tool for each part of the job. Syphon gives you flexibility to break out of single-app solutions and mix creative applications to suit your needs.*

So let's connect to the Internet and get both components that we will need for this project. We will start with getting the Syphon implementation for Cinder.

Before you begin, make sure that you are on Mac OS X and that your operating system version is 10.6 (Snow Leopard) or later. Syphon won't work with older versions of the Mac OS X.

We will need to download the source code from the Syphon implementation's Google Code repository. To do that, make a directory somewhere on your computer and call it `syphon`. Open the `Terminal` application (found in `Applications/Utilities/Terminal`) and navigate to it by using the `cd` command:

```
cd /your/path/to/syphon
```

Hit *Return* and then type:

```
svn checkout http://syphon-implementations.googlecode.com/svn/trunk/
syphon-implementations-read-only
```

Hit *Return* again and wait. This will download the latest source code of all Syphon implementations to your computer.

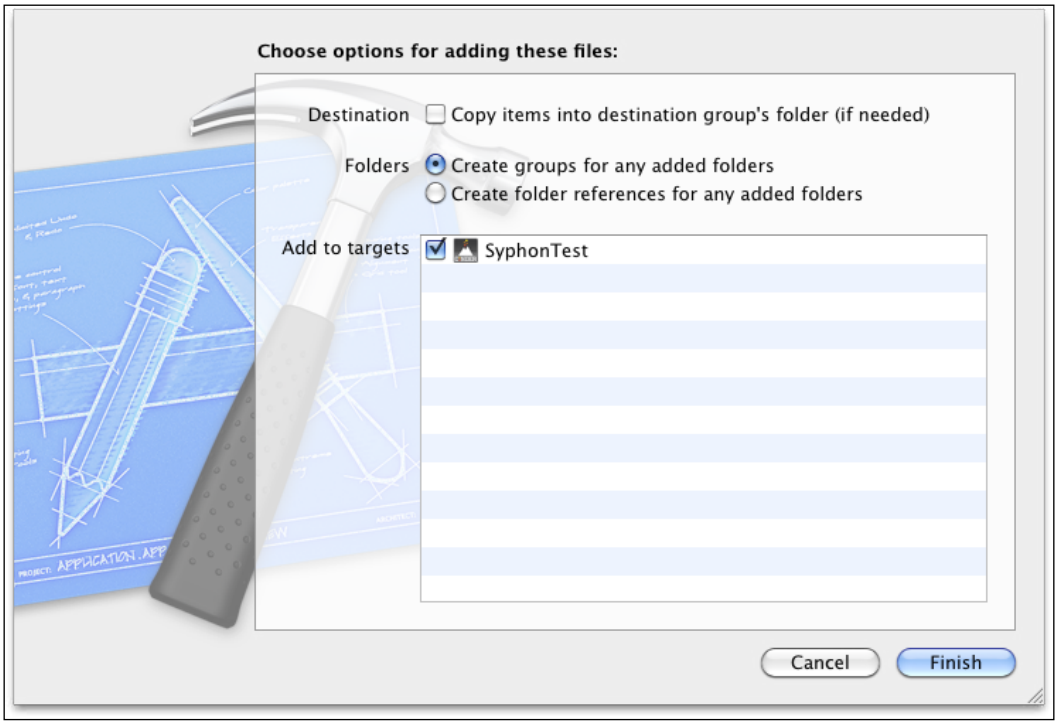
Another thing that we will need is MadMapper. Go to `http://www.madmapper.com/` and download the demo version. The demo version will have all the functionality that we will need in this project, except we won't be able to save the project and we will see a floating watermark in the final output.

Creating a visual counter application in Cinder

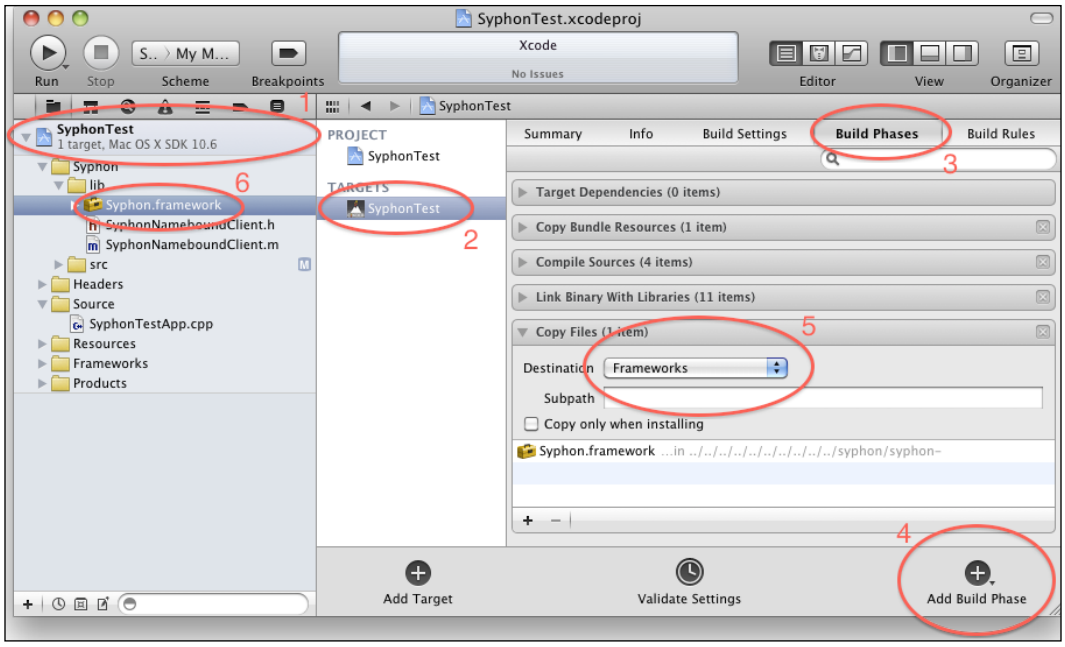
Next, create a new project by using TinderBox. Call it `SyphonTest` and open the Xcode project file, `xcode/SyphonTest.xcodeproj`.

To make Syphon work with our app, we need to include the source files. Navigate to the directory where you downloaded all the Syphon implementations and go to the folder `syphon-implementations-read-only/Syphon Implementations/Cinder/ciSyphon/blocks`.

You will see another Syphon directory inside it. Click and drag it to our Xcode project window and drop it on the **SyphonTest** Xcode project icon in the project navigator pane. Xcode will ask you to define several import options. Make sure that **SyphonTest** in the **Add To Targets** part is checked. Other options are up to you, but I prefer to leave them default for this project. Click on **Finish**.



Now, the Syphon header and implementation source files and the framework itself are added to our project. There is one more thing that has to be done before we begin; we need to make sure that `Syphon.framework` is copied into the application bundle, otherwise it won't work.



To do that, we need to add a **Copy Files** build phase to the application compile process.

1. Click on the **SyphonTest** Xcode project icon in the project navigator.
2. Select **SyphonTest** in the **Targets** section.
3. Go to the **Build Phases** tab.
4. Add a new build phase by navigating to **Add Build Phase | Add Copy Files**.
5. Choose **Frameworks** in the **Destination** field.
6. Drag-and-drop `Syphon.framework` from the `Syphon/lib` directory.

You should be able to compile the application now, but it is possible that you will get the following error while trying to compile it:

```
No member named 'getTextureId' in 'cinder::gl::Texture'
```

Consider the case when Cinder is in development all the time and some method names are changing time by time. In this case, the `getTextureId()` method in the `Texture` class has turned into `getId()`. We can fix it by opening up the `syphonServer.mm` file under the `Syphon/src` directory in the project navigator and changing the following line in the `publishTexture()` method:

```
GLuint texID = inputTexture->getTextureId();
```

To the following one:

```
GLuint texID = inputTexture->getId();
```

With this fixed, our project should compile and run now.

Next we are going to draw something on the screen. As we did not manage to do anything with text in the previous chapters, let's do it now. Cinder has very good text support in terms that it supports unicode characters. We won't use them in this example, but keep this in mind in case you need it.

To make things work, we will need to add a couple of headers. Add the following lines to the header part of our code:

```
#include "cinder/Text.h"
#include "cinder/Font.h"
#include "cinder/gl/Texture.h"
#include "cinder/Utilities.h"
#include "cinderSyphon.h"
```

As you can see, we will need the `Text`, `Font`, and `Texture` features in our project as well as a handy function for converting numbers to strings (the `toString()` function), and of course, Syphon.

Next, we have to declare some variables! Go to the class declaration and add the following lines:

```
TextBox textBox;
gl::Texture texture;
int counter;
syphonServer syphon;
```

As you can see, we are creating a `textBox`, a `texture`, a `counter`, and a `syphon` server. `textBox` is going to be rendered inside `texture` and the `textBox` contents are going to be created by using the `counter` variable. Afterwards, we are going to publish the final image to the `syphon` server so that other applications can access it.

Now we have to start writing some real code. Let's navigate ourselves to the `setup()` method implementation and add the following code there (you can skip the comments):

```
// set the initial count of the counter
counter = 0;

// define text box text color
textBox.setColor( Color( 0.7f, 0.7f, 0.7f ) );
```

```
// set font for the text
// you can chose any font that is installed on your system
textBox.setFont( Font( "Arial-Black", 200 ) );

// this will print out the names of all available fonts
// it can help in case you are not sure what is the
// exact name of the font you want to chose
vector<string>::const_iterator fName;
for( fName = Font::getNames().begin(); fName != Font::getNames().
end(); ++fName ) {
  console() << *fName << endl;
}

// set text alignment within the text box,
// TextBox::LEFT, TextBox::RIGHT are also available
textBox.setAlignment( TextBox::CENTER );

// set the width of the text box to the width of the window
// and height to TextBox::GROW, so it takes the height
// of the contents
textBox.setSize( Vec2f(getWindowWidth(),TextBox::GROW) );

// turn on smooth rendering of the text box
textBox.setPremultiplied( true );

// set the syphon server name
syphon.setName("Counter");
```

This is the set up for the basic properties of the text box and it initializes the `counter` variable, which is a kind of core of our application.

For the next part, let's scroll down to the `update()` method implementation and add the following lines of code there:

```
// convert counter int value to string
string text = toString(counter);

// assign text to our text box
textBox.setText( text );

// render our text box into texture
texture = gl::Texture( textBox.render() );

// increment counter
counter++;
```

This code will prepare the current frame for drawing in terms of setting new text for the text box, rendering it into a `Texture` object, and increasing the counter by one. We are using the `toString()` function to convert the counter `int` value to a `string` type.

Finally, we go to the `draw()` method implementation and add our drawing code just after the `gl::clear()` call:

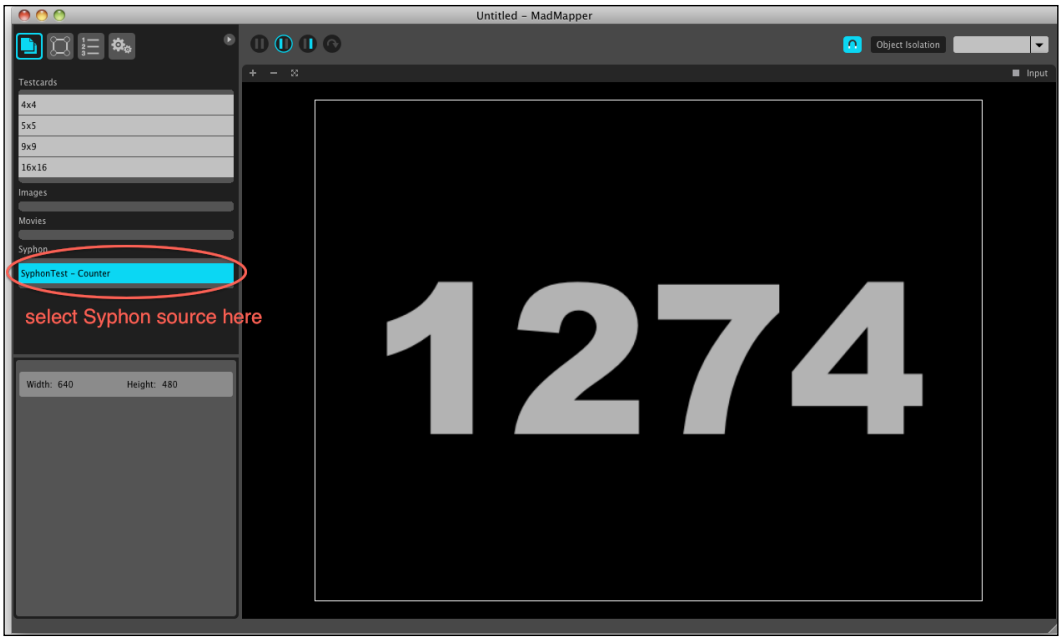
```
// check if our texture is prepared
if ( texture ) {
    // calculate the position of the textBox texture
    Vec2f pos = ( getWindowSize() -
        Vec2f( textBox.getSize().x, textBox.measure().y ) ) / 2;

    // draw text texture on the screen
    gl::draw( texture, pos );
}

// publish screen to the syphon server
syphon.publishScreen();
```

We check the texture before we draw it to avoid application crash. A Cinder app will crash if we try to draw an empty texture. Another strange thing might be the way we calculate the position of our `textBox` texture—we use the `getSize()` method of `textBox` to get its width and the `textBox.measure()` method to get its height. Why is that? I think that this part of Cinder will be improved over time, but right now it's just that we will get `0` as the height of the textbox if we call `textBox.getSize()`. It will work if we define a fixed textbox height, but as for now the height is defined as growing or flexible, so it does not. On the other hand, we can get the height of the contents by calling `textBox.measure()`. You have to play around with these two and the `setSize()` method to understand it more clearly, but this solution will work for now.

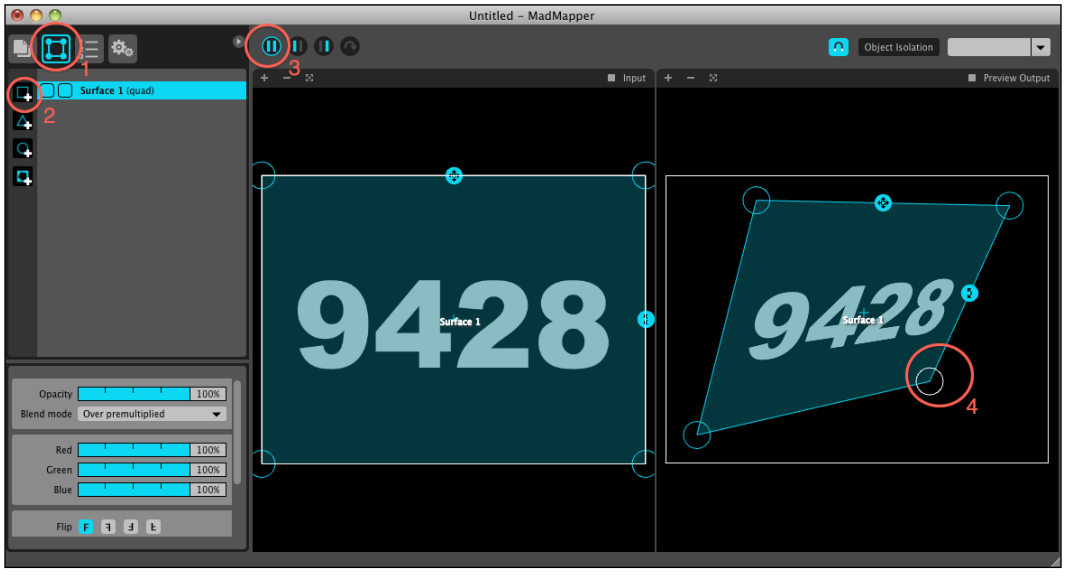
The last line of the code we just added should be self-explanatory—we publish the whole screen to our Syphon server. Compile and run the application, so we can test if it is actually working.

Using a Cinder application as source in MadMapper

Open MadMapper and make sure that our application is still running. You should see **SyphonTest – Counter** as Syphon source in MadMapper:



Double-click on it to activate. You can start to adjust the image to a surface. A MadMapper tutorial is outside the scope of this book, but the following screenshot should explain the very basics of the process:



Follow these steps (numbers in the preceding screenshot):

1. Go to the surfaces panel by clicking on the icon.
2. Add a rectangular surface.
3. Switch to the split-view layout.
4. Drag the handles to map out the image.

This is it. We have gained some basic understanding of how to create creative visual applications for using them together with other applications of a similar kind. Visit `http://www.madmapper.com/` to learn more about MadMapper and `http://syphon.v002.info/` to learn more about Syphon. To learn more about their use with Cinder, take a look in the Cinder forums.

Using OSC and Cinder

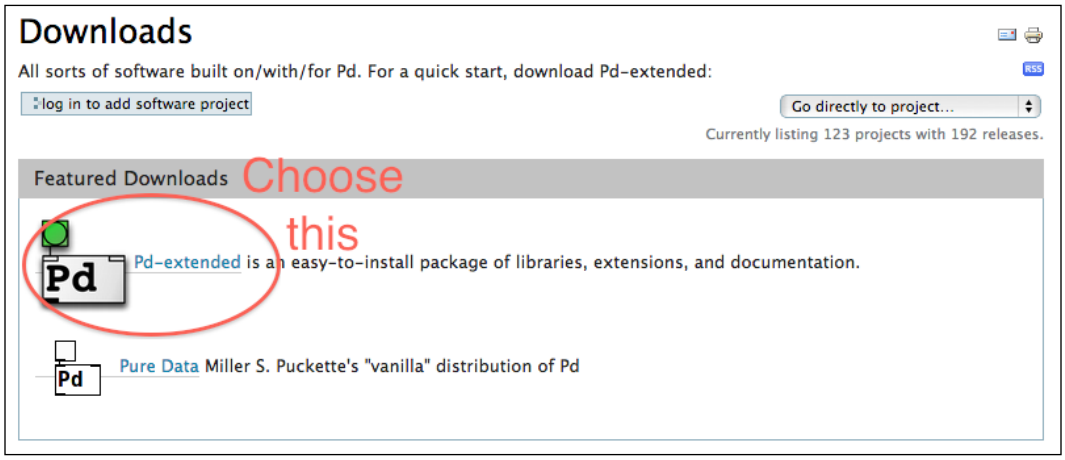
Another project that we will create is going to connect a visual Cinder application with a sound application that is created in Pure Data visual programming language. If you don't know Pure Data already, it is a programming environment where you place different kinds of boxes on the canvas and connect them with wires instead of writing lines of code in a text or code editor.

The sound and visual applications will communicate by sending different parameters back and forth between each other. To do that, we will use the **open sound control** (**OSC**) content format.

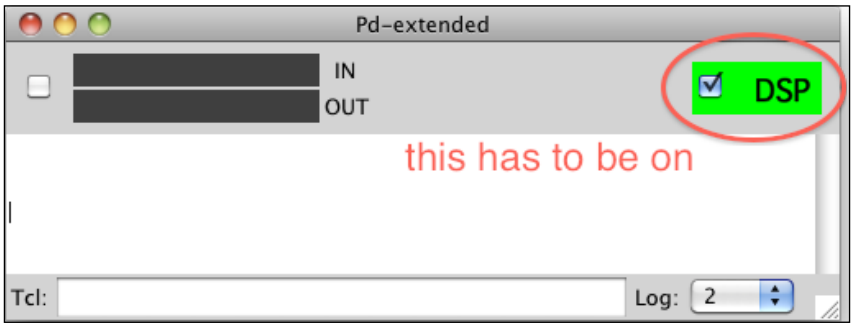
Creating a Pure Data audio application

Before we begin to work on the Cinder part of it, we have to get Pure Data and be able to run it. Go ahead and point your browser to the official Pure Data website (`http://puredata.info/`).

Go to the **Downloads** section and choose the **PD-extended** version as it contains all the plugins we need for this project.

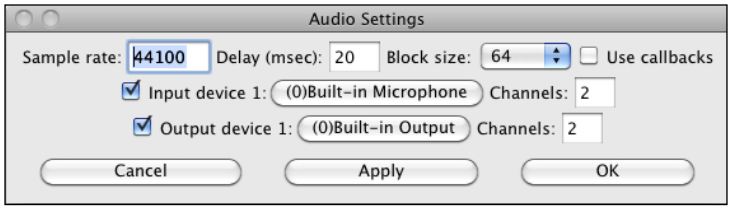


Install and open it. You should see a console:



Notice the **DSP** checkbox/button. It has to be checked to be able to hear some sound that is made with Pure Data. By default it is off, so turn it on by checking the checkbox.

You have to make sure that your audio output is configured properly, so go to Pure Data audio preferences (**Media | Audio Settings** on Mac OS X) and make sure that they look approximately as shown in the following screenshot:



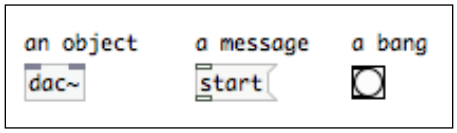
The most important parts here are the sample rate and the **Output device 1**. **44100** is the default CD quality sample rate and for now we are ready to go with **Built-in Output**. If you prefer to use some kind of external sound card, you might want to select your sound card output as the main output device.

Go to **File | New** to create a new patch. Pure Data documents are called **patches**, so don't be confused when you see this strange term from now on.

A fresh, blank window should be open in front of you right now. As mentioned before, to make a program in Pure Data, one has to place boxes and connect them with wires. So, let's create a box by choosing **Put | Object** from the menu bar. Place it on the canvas by positioning it with the mouse and then confirm it by hitting any key on the keyboard.

There are many types of boxes that you can create in Pure Data, but it is outside the scope of this book to explain all of them. For now we will make use of the following:

- Object
- Message
- Bang

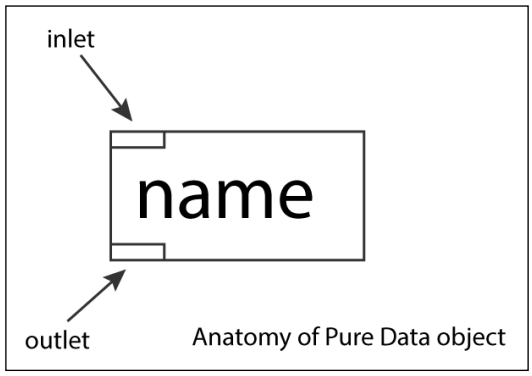


An object is the same as a class instance. You instantiate it by placing it on the canvas and call methods by sending messages.

A message can be considered as an instance method call. You need to specify the name of the method and parameters that have to be passed.

A bang is something that does not have such a direct similarity to something in the world of object-oriented programming. Nevertheless, it can be seen as an impulse that makes a method call possible. A bang can be triggered by a user (by clicking on it) or by the program itself (connecting some kind of outlet to the bang inlet).

Each object has inlets and outlets (the rectangular squares on the top and bottom of an object). To make the program work, we need to connect different outlets to the appropriate inlets and the connections can be considered as pipes where the data is being pushed from one object to another.



An object can have multiple inlets and outlets.

So, let's place an object on the canvas by choosing **Put** | **Object** from the menu bar. Click somewhere on the canvas and write `loadbang` in the box. This is a specific object in Pure Data that executes a bang once the patch is loaded (opened). This can be very useful if you are working on multiple machines where you have a little control over the working environment and can help you automatically prepare the Pure Data environment for your needs.

So, let's do a little trick and place a message on the canvas by choosing **Put** | **Message** from the menu bar. Enter the following code in the box:

```
;
pd dsp 1
```

Connect the outlet of the `loadbang` object to the inlet of the message so that it looks as follows:

This will trigger the DSP button, as discussed earlier, to checked every time you open your patch. Save the file at some safe location.

We are going to create a simple audio sample playback application that will load and play back sound on demand. Find some short audio samples in `wav` format (mine are `kick.wav` and `snare.wav` found on `http://www.freesound.org/`) and place them in the same directory where you saved your Pure Data patch.

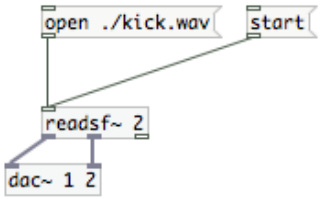
We are going to use the `readsf~` object to open and play the sound files. So, let's place a new object on the canvas by choosing **Put | Object** from the menu bar. Write `readsf~` in the box. To make the object do something, we have to send messages to it. We will need to send two messages—open and send. So, go on and put them on the canvas. Chose **Put | Message** from the menu bar, place the message on the canvas, and write:

```
open ./kick.wav
```

Place another message and write:

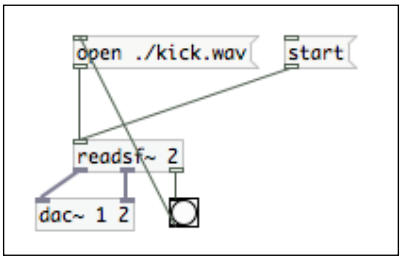
```
start
```

Connect both messages to the `readsf~` object. We won't hear any sound by now—we have to place a `dac~` object on the canvas to make it happen. `dac~` stands for digital to analog converter. You need your computer's sound card to keep it simple. So, create a `dac~` object by choosing **Put | Object** from the menu bar again. Add parameters 1 and 2 so that we gain access to both channels for proper stereo output. You should add a parameter 2 to the `readsf~` object as well so that it appears as follows:



Let's test the setup by entering the run mode. You can do that either by choosing **Edit | Edit Mode** from the menu bar or simply using a keyboard shortcut *Cmd + E*. Notice that the message boxes act as buttons now. Notice that nothing happens if you click on the `start` message right away. That is because you have to load the sound file before you start playing it. So, click on the `open` message and then again on the `start`. You should hear a sound now.

If you press `start` again—nothing happens—you have to load the sound again before playing it. We can automate this process by triggering the `open` message, when the sound has finished playing. Luckily, this is very easy to do as the last outlet of the `readsf~` object outputs a bang once the end of the audio file has been reached. We could connect the outlet to the `open` message directly, but to make this more obvious, let's put a bang object in-between so that we can actually see when the bang happens. So, go back to the edit mode and select **Put | Bang** from the menu bar to place a bang on the canvas and connect the nodes as shown in the following screenshot:

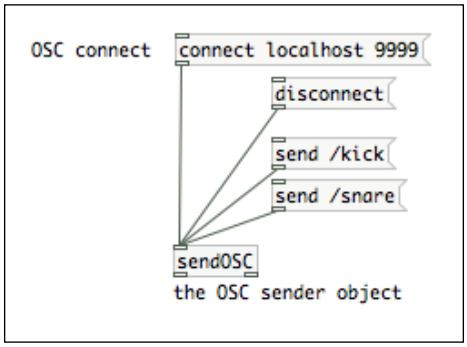


Now, enter the run mode again by pressing *Cmd + E* and click on the `open` and `start` messages sequentially. You should mention that the bang object blinks when the playback is finished. Now, you don't need to press the `open` message again, just the `start` one from now on.

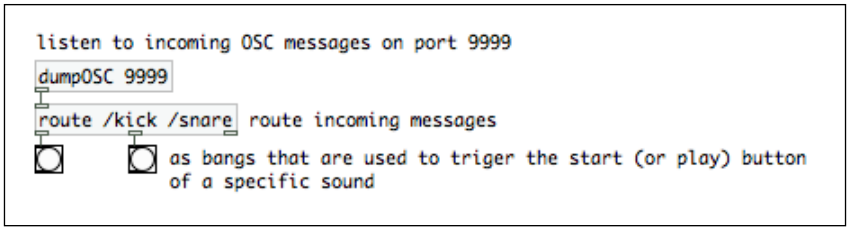
To make things more automatic, connect the `loadbang` object outlet to the `open` message; the sound file will be loaded automatically as the patch is loaded.

Adding OSC to a Pure Data patch

The next thing we need to implement is the networking with OSC. We will be using two objects for that—`sendOSC` and `dumpOSC`. Go on and add the `sendOSC` object and messages that you can see in the following screenshot to the patch:



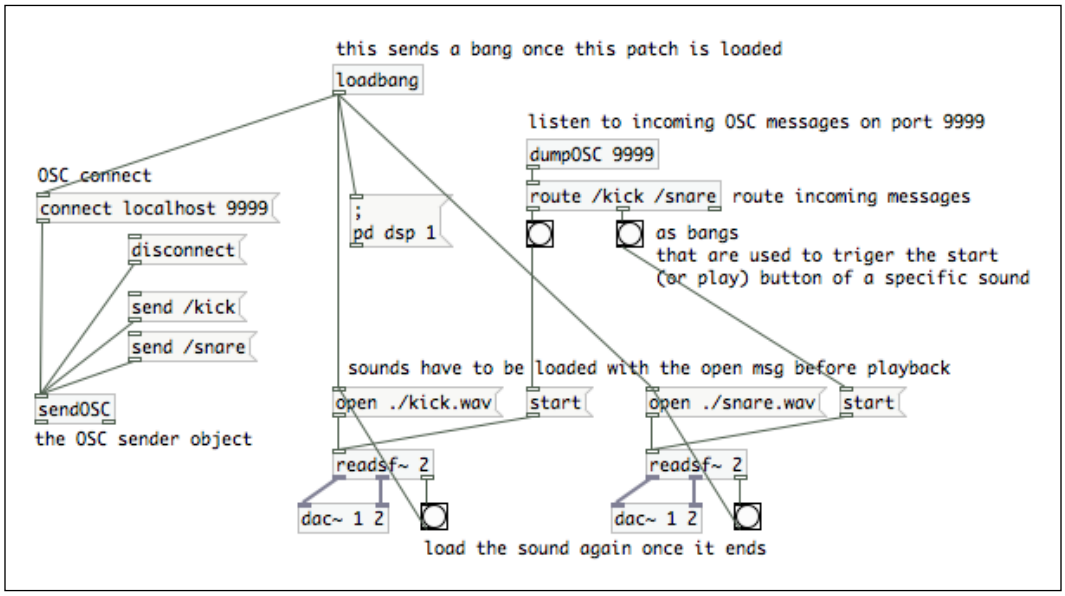
This will allow us to send OSC messages `/kick` and `/snare`. We need some more to be able to receive these messages so that we can test if it actually works. Add the following to the patch:



So, we can test the OSC connection now by entering the run mode (*Cmd* + *E*) and clicking on the `connect` message. This will set up the OSC sender object to send messages to localhost (`127.0.0.1` is the local network address of your own computer even if you are not connected to a network) port 9999. Now, when you click on the `send /kick` or `send /snare` messages, the bang objects under the `dumpOSC` object should blink.

The route object takes any input and checks if it equals one of the space-separated parameters (in this case, `/kick` and `/snare`). If it does, a bang is triggered from the according outlet—first is for the first parameter, second for the second, and so on. You should click on the `disconnect` message, when the job is done.

Next we connect the bang objects to the `start` messages of the `readsf~` object. Yes, we don't have an object for the snare sound yet. As there is nothing new to it, just take a look at the full patch:

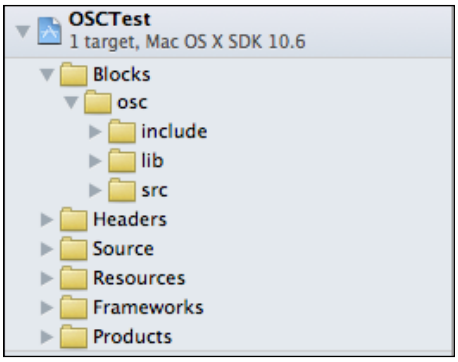


The Pure Data part of our project is ready. Next, we have to create a Cinder application that can send and receive OSC messages.

Creating an OSC application in Cinder

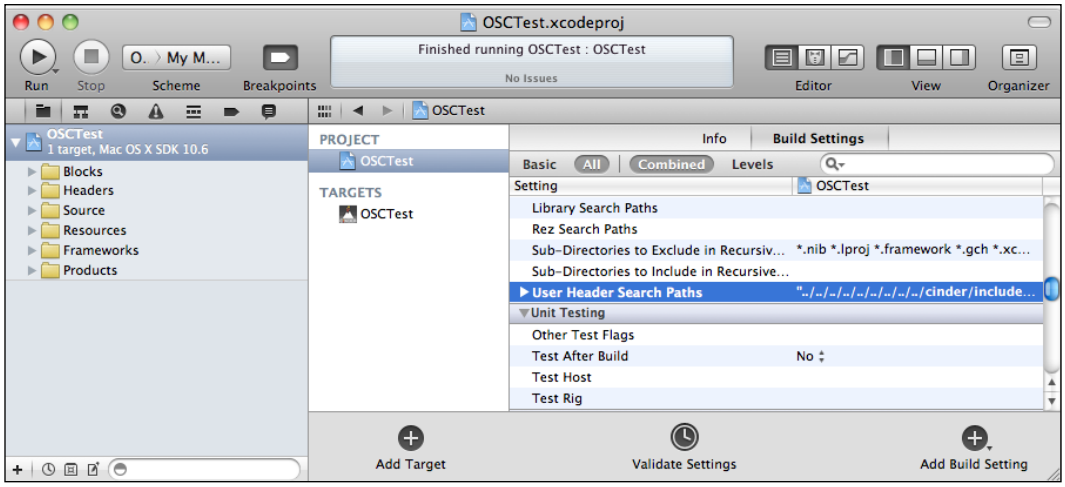
Let's use TinderBox to create a new Cinder project and let's call it `OSCTest`. Go to the project directory and open the project file (`xcode/OSCTest.xcodeproj` on Mac OS X or `vc10/OSCTest.sln` on Windows). Open `OSCTestApp.cpp` in the editor.

We need to add the OSC block to the project source to be able to use OSC messaging. In your main Cinder directory, find a folder named `blocks`. Drag the `osc` folder from there to the project navigator. Delete the `samples` folder, so the project navigator appears as follows:



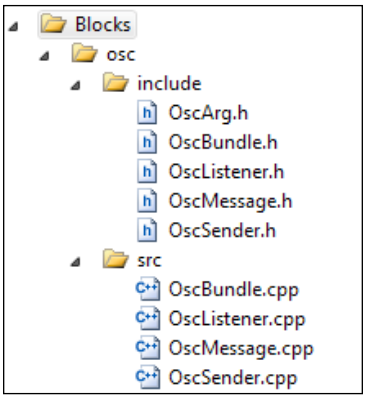
In case you are using the AppRewrite version of Cinder, TinderBox can add the OSC block to the project for you.

If you try to compile now, you will see that it is not possible. We have to make some changes in the project's build settings.



Add a new entry `$(CINDER_PATH)/blocks/osc/include` to the **User Header Search Paths** field:

It is a bit of a different story with Windows though. First, you have to create the following folder structure by yourself (right-click and navigate to **Add** | **New Filter**) and then add files from the OSC block (located under the `cinder\blocks` directory) `include` and the `src` directories by dragging them on the appropriate folders.



Then, we have to open the **Project Properties** window by choosing **Project** | **OSCTest Properties** from the menu bar, and add `Path-to-cinder\blocks\osc\include` to **C/C++** | **General** | **Additional Include Directories**, `Path-to-cinder\blocks\osc\lib\vc10` to **Linker** | **General** | **Additional Library Directories**, and `osc_d.lib` to the **Linker** | **Input** | **Additional Dependencies**.

There should be no errors while building and running the application now.

Next, we have to build the actual application that is supposed to communicate with our Pure Data patch. First, we are going to send a `/kick` message on left mouse button click and a `/snare` message on the right one. Later, we will add some more code that will allow us to listen to these messages and draw something on the screen.

To get started with the code part, we need to include some headers:

```
#include "OscSender.h"
#include "OscListener.h"
#include "OscMessage.h"
```

As you may have guessed, `OscSender.h` holds all the necessary code for sending OSC messages, `OscListener.h` holds the code for listening to the messages, and `OscMessage.h` holds the code for constructing a new message and parsing an incoming OSC messages.

We need an `OscSender` instance, so let's go to the class declaration part and add the following line of code there:

```
osc::Sender sender;
```

Next, navigate to the `setup()` method implementation and add the following code there:

```
sender.setup("localhost", 9999);
```

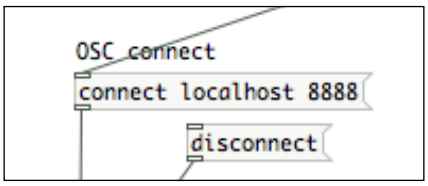
This will set up `OscSender` to send messages to localhost port 9999—the same parameters we use in the Pure Data patch.

Next, we have to use the `mouseDown()` method, so let's navigate to its implementation and add the following code:

```
osc::Message message;
if ( event.isLeft() ) {
    message.setAddress("/kick");
} else {
    message.setAddress("/snare");
}
sender.sendMessage(message);
```

Here, we tell the application that it has to send a `/kick` message, if the user presses left mouse button, or a `/snare` message, if any other mouse button is pressed on the Cinder application window. Compile and run to test if it actually works. Make sure that our Pure Data patch is open and running.

Next, we have to make our Cinder application respond to the same `/kick` and `/snare` messages. The most important thing here is that we need to remember that there can't be two OSC listeners that are listening to the same port on the same machine, but there can be multiple senders though. So, to receive messages from the Pure Data patch, we will need to change the parameters of the `connect` message. Let's change the port to 8888 instead of 9999:



Now we are ready to listen to some messages from Pure Data. We have to add two new variables in the class declaration:

```
osc::Listener listener;
string lastMessage;
```

We will use the `lastMessage` variable to store the last OSC message we received and draw a shape depending on it.

The listener has to be set up, so let's do it by adding the following line of code in the `setup()` method:

```
listener.setup(8888);
```

You can see here that this listener will listen to the port 8888, that is, the same port number we defined for the OSC sender object in the Pure Data patch.

Next, we have to add some code that actually reads the messages from the listener. Let's go to the `update()` method implementation and add the following code there:

```
while ( listener.hasWaitingMessages() ) {
    osc::Message message;
    listener.getNextMessage( &message );
    lastMessage = message.getAddress();
}
```

This `while` loop gets all the waiting messages in the OSC listener's buffer by reading and analyzing them one by one. To use the `listener.getNextMessage()` method, we have to pass a reference to a message object—`osc::Message message`. The `listener.getNextMessage()` method fills the message by accessing it through its reference `&message`. Finally, we assign the `address` parameter of the message to the `lastMessage` variable.

Now, we can move to the `draw()` method implementation and draw different shapes for each message. Add the following code there to make it happen:

```
if ( lastMessage == "/kick" ) {
    gl::drawSolidCircle(getWindowCenter(), 100, 3);
}

if ( lastMessage == "/snare" ) {
    gl::drawSolidCircle(getWindowCenter(), 50);
}
```

Now, we are ready to compile and run the application. Do it and switch over to the Pure Data patch. Before you send any messages, disconnect with the port 9999 to destroy the previous session and make a new connection with port 8888 by clicking on the `connect` message. Click on the `/kick` and `/snare` messages, and see what's happening in our Cinder application window. It should draw a triangle when we send a `/kick` message and a circle when we send a `/snare`.

So, these are the very basics of OSC communication with Cinder. The best thing about it is that you can use it over local or global network—just use real IP addresses or even domain names instead of localhost.

Summary

In this chapter, we went over two advanced uses of Cinder together with other applications on the same and other possibly networked computers. We learned about the Syphon framework and how to use it to send a moving image from one application to another without any delay. We learned the basics of the MadMapper projection mapping software that can prove itself very useful, if you want to use a 3D object as a screen for your generative Cinder projection. Another thing that we understood is how to send messages from one application to another and back. We gained some introductory knowledge about the Pure Data visual programming environment that can be useful if you want to synchronize generative sound with generative visuals on the same or separate (but networked) computers. With this kind of basic knowledge, you don't have almost any limits to unleash your creativity towards creative coding at all!