

Interactive Tables

Edouard Bouvet

July 17, 2013

1 Introduction

1.1 About

This framework is supposed to simplify the development of applications on multitouch devices such as interactive tables or tablets.

The framework is entirely based on JavaScript. It uses the TUIO protocol for collecting the multitouch events but can easily be adapted to work with other devices such as tablets or phones. It contains a network module that allow you to create applications which can communicate with other multitouch devices. It also contains a 2D rigid body simulation engine designed for fully dynamical applications. A gesture recognition library is also integrated in the framework.

As the framework is written in JavaScript, it is supposed to work with most of the actual browsers. Other JavaScript libraries can also easily be added to the framework.

Some examples for tables and tablets are also provided with the framework. Be sure to look at them since a lot of the framework mechanics are illustrated there.

You will find all the examples shown in this manual in the doc folder.

1.2 Prerequisites

Some knowledge of web development are required to use the framework. It is important to understand how HTML and JavaScript interact on a web page. As the framework is only mainly based on the HTML5 Canvas, it is not very important to know all the HTML tags. You'll also probably have to learn how OOP and inheritance mechanics work in JavaScript.

You can easily find some good tutorials on google. Here are some examples :

- Some basic HTML5 examples : http://www.w3schools.com/html/html_examples.asp
- Some basic JavaScript examples : http://www.w3schools.com/js/js_examples.asp
- Lots of HTML5/JavaScript ressources : <http://www.html5rocks.com/en/>

- Introduction to JavaScript inheritance mechanics : <http://phrogz.net/JS/classes/OOPinJS2.html>

You can also look at the Tuio specifications to understand how the protocol works (<http://www.tuio.org/?specification>).

1.3 Working environment

You will need a TUIO Client to use the framework. The client needs to work with the 1.1 version of the TUIO protocol so it can generate blobs. Comunity Core Vision 1.4 or above will work fine when you change the protocol version in the *"app_settings.xml"* file.

Since the TUIO protocol only provides UDP messages and since a web browser only accept TCP messages, we need to forward the messages. A node.js port and protocol forwarder is provided with the framework. You only need to install node.js (<http://nodejs.org/>), then open a console, get to the directory and execute the commande : *"node server.js"*.

Be careful when choosing a browser to open your application. Not all browsers support every functionalities of JavaScript. Google Chrome and Firefox work most of the time on desktop. On mobile devices, you can check that table to make a choice : <http://mobilehtml5.org/>.

If you plan to create networking applications, you will need a web server. If it is supposed to run on a local network, you can use <http://sourceforge.net/projects/xampp/> or <http://www.easyphp.org/>.

1.4 Concept

Manager : This is the main class of the framework. From this class you will be able to access all of your shapes, patterns and blobs. This is also the class that manage the physics engine and create the gesture event.

Rectangle : This class allow the user to add rectangle to its application. It is possible to add physics to the rectangle.

Circle : This class allow the user to add circle to its application. It is possible to add physics to the circle.

Pattern : Patterns are usefull to recognize different types of blobs. You can define a pattern following the size of the blob, its angle or its position.

Tuio: This is the TUIO layer. TUIO client and blobs are defined here.

You can easily use inheritance on those classes for your projects, as shown in the *"Light"* example.

2 Basics

2.1 Minimal code

The minimal code starts with a lot of inclusion. Please notice that the HTML5 canvas are created at the beginning of the body section.

```
// The width and height attributes are not really important since they
// will be modified later.
// The left and top attributes determine where the canvas will start.
// The z-index is the position of the layer. A canvas with a z-index of
// 1 would be drawn on top of a canvas of a z-index of 0.
<canvas id="canvas" width="1680" height="645" style="position: absolute;
left: 0; top: 0; z-index: 0;"></canvas>
```

At the beginning of the script we can find some global declarations.

```
// Initialize the gesture recognition plugin.
Hammer.plugins.fakeMultitouch();
// Create a TUIO Client listening to the port 5000 of localhost
var client = new Tuio.Client(
host: "http://localhost:5000"
});
// Get the screen width and height
var screenW = $(window).width();
var screenH = $(window).height();
// Create the manager
var manager = new Manager();
// Create the javascript canvas and context
var canvas = null;
var context = null;
```

Then we can find the main functions.

```
// Gesture recognition function
gesture = function (event) {
    //All the processing relative to the gesture can be written here.
};

// Init function
onConnect = function () {
    // Canvas Init
    // link the javascript canvas to the HTML5 canvas
    canvas = $("#canvas").get(0);
    // set the size of the canvas to fullscreen
    canvas.width = screenW;
    canvas.height = screenH;
    // get the context of the canvas
    context = canvas.getContext("2d");
```

```

// Manager Init
manager.setScreenSize(screenW,screenH);
manager.setOnGesture(gesture);

// Fill the screen with black
context.fillStyle = "#000000";
context.fillRect(0, 0, canvas.width, canvas.height);
};

// Draw the objects
draw = function () {
// All the processing that need to be executed on each frame will be
  here.
};

```

The last part contains the function triggered by the Tuio events.

```

// Refresh the screen
onRefresh = function (time) {
  draw();
};

// New blob event
onAddTuioBlob = function (blb) {
  manager.addEvent(blb);
};

// Removed blob event
onRemoveTuioBlob = function (blb) {
  manager.removeEvent(blb);
};

// Updated blob event
onUpdateTuioBlob = function (blb) {
  manager.updateEvent(blb);
};

// Make the link between Tuio event and functions
client.on("connect", onConnect);
client.on("addTuioBlob", onAddTuioBlob);
client.on("updateTuioBlob",onUpdateTuioBlob);
client.on("removeTuioBlob",onRemoveTuioBlob);
client.on("refresh",onRefresh);
client.connect();

```

If you open the page with a browser, you will only see a black screen.

2.2 Adding a shape

In this chapter we will see how to draw a shape on the canvas.

There are two ways on positioning a shape.

You can set absolute positions. This means that no matters where the page

is open, your shape will always be at the position X,Y and measures Z pixels. With that way, if the page is not open on the screen that it was made for, it will often be ugly or incomplete.

The other way requires to set position relatively to the canvas size. If you want your shape to be in the middle of the screen, whatever the screen is, it is possible.

For example, we can add a new shape:

```
var rect = new Rectangle();
```

Then we just set the shape attributes :

```
// Init function
onConnect = function() {
    // Canvas Init
    canvas = $("#canvas").get(0);
    canvas.width=screenW;
    canvas.height=screenH;
    context = canvas.getContext("2d");

    // Manager Init
    manager.setScreenSize(screenW, screenH);
    manager.setOnGesture(gesture);

    // Add a new shape
    // Set the size of the canvas of the shape
    rect.setCanvasSize(canvas.width, canvas.height);
    // Set the context where the shape have to be drawn
    rect.setContext(context);
    // Set the shape position
    rect.setXRelativePosition(0.5);
    rect.setYRelativePosition(0.5);
    // Set the shape size
    rect.setRelativeHeight(0.3);
    rect.setRelativeWidth(0.5);
    // Set the color of the shape
    rect.setColor("#FFFF00");
    // Give the shape to the manager
    manager.addRectangle(rect);

    // Fill the screen with black
    context.fillStyle = "#000000";
    context.fillRect(0, 0, canvas.width, canvas.height);
};
```

Finally we tell the manager to draw the rectangle on the canvas :

```
// Each time we refresh the screen, we redraw the rectangle
draw = function () {
    manager.drawRectangle();
};
```

2.3 MultiCanvas

Using this framework, you can make applications with several canvas. It is possible to put them side by side or on top of each other.

An interesting use of the canvas is to have at least two of them, one for the background and one for the foreground of your application. On the background canvas, you will draw everything that is not supposed to move in your application. This canvas will be drawn only once at the beginning of the application. On the foreground canvas, you will draw every other objects. This method decreases the processing time at each frame and give you better performances.

In this example, we will move a ball at each frame. We will add another canvas to draw the ball.

Don't forget to add the new HTML5 canvas :

```
// Notice that the z-index of the two canvas is different
<canvas id="canvasForeground" width="1680" height="645" style="position:
    absolute; left: 0; top: 0; z-index: 1;"></canvas>
<canvas id="canvasBackground" width="1680" height="645" style="position:
    absolute; left: 0; top: 0; z-index: 0;"></canvas>
```

We will now add a new circle and the javascript canvas and contexts :

```
var canvasForeground = null;
var contextForeground = null;
var canvasBackground = null;
var contextBackground = null;
var ball = new circle();
```

We have to initialize the new canvas, contexts and the ball

```
// Init function
onConnect = function() {
    // Canvas Init
    canvasForeground = $("#canvasForeground").get(0);
    canvasForeground.width=screenW;
    canvasForeground.height=screenH;
    contextForeground = canvasForeground.getContext("2d");

    canvasBackground = $("#canvasBackground").get(0);
    canvasBackground.width=screenW;
    canvasBackground.height=screenH;
    contextBackground = canvasBackground.getContext("2d");

    // Manager Init
    manager.setScreenSize(screenW, screenH);
    manager.setOnGesture(gesture);

    // Add a new shape
    ball.setCanvasSize(canvasForeground.width, canvasForeground.height);
    ball.setContext(contextForeground);
    // We put the ball on the left side of the screen
    ball.setXRelativePosition(0);
```

```

    ball.setYRelativePosition(0.5);
    ball.setRelativeRadius(0.03);
    ball.setColor("#FFFF00");
    manager.addCircle(ball);

    // Fill the screen with black
    contextBackground.fillStyle = "#000000";
    contextBackground.fillRect(0, 0, canvasBackground.width,
                               canvasBackground.height);
};

```

Now we change the draw function :

```

// Draw the objects
draw = function() {
    // On each frame, we clear the last drawings on the foreground
    contextForeground.clearRect(0, 0, canvasForeground.width,
                                canvasForeground.height);
    // We get the actual position of the ball and then move it a bit on
    // the right
    var xPos = ball.getXAbsolutePosition();
    ball.setXAbsolutePosition(xPos + 1);
    //We then draw the ball at its new position
    manager.drawCircle();
};

```

3 Touch Events

3.1 Blobs

The TUIO protocol send messages about the states of the blobs at regular intervals. Then we call the draw function for each message received. If we want to catch all the touch events, we just have to get them in the draw function. All the processing can be there too. We also have to define a default pattern. We will see in the next chapter what it is exactly.

```

var def = new Pattern();

```

We had to add the pattern to the manager.

```

// Init function
onConnect = function() {
    // Canvas Init
    .
    .
    .

    // Manager Init
    manager.setScreenSize(screenW, screenH);
    manager.setOnGesture(gesture);
    manager.addPattern(def);
};

```

```

    .
    .
    .
};

```

And then we just get the blobs and draw them

```

draw = function() {
    contextForeground.clearRect(0, 0, canvasForeground.width,
        canvasForeground.height);
    // Get the blobs
    var blobs = client.getTuioBlobs();
    // Processing - Don't forget to set a context if you want to draw
    // the blobs.
    for (var i in blobs) {
        blobs[i].setContext(contextForeground);
    }
    // Draw the blobs
    manager.drawBlobs(blobs);
};

```

3.2 Patterns

Since some multitouch devices can recognize other objects than fingers, you will probably want to be able to know what kind of object was in contact with your device. For that, it is possible to define patterns. You can set limits for the patterns on the size of the object, the angle or the position. /* If we want to make a difference between an object with a known size and other objects, we will define a pattern for the known object. We will keep the default pattern to recognize the other objects.

```

var obj = new Pattern();
var def = new Pattern();

```

Then we will have to set the limits for the patterns.

```

// Init function
onConnect = function() {
    // Canvas Init
    .
    .
    .

    // Pattern Init
    // Set a size limit of 500 + or - 200 on the pattern
    obj.addSizeLimit(500,200);
    // Add the patterns to the manager
    manager.addPattern(obj);
    manager.addPattern(def);
    // Manager Init
    .
    .
    .
}

```



```
};
```

And finally we can associate a blob to the better matching pattern and do the processing.

```
// Draw the objects
draw = function() {
    contextForeground.clearRect(0, 0, canvasForeground.width,
        canvasForeground.height);
    var blobs = client.getTuioBlobs();

    for (var i in blobs) {
        // Set a context for all the blobs
        blobs[i].setContext(contextForeground);
        // If the blob match with the obj pattern, we set the color to
        // green
        if (manager.blobMatchPattern(blobs[i],obj)) {
            blobs[i].setColor("#00FF00");
        } else {
            // If the blob doesn't match, we set the color to yellow
            blobs[i].setColot("#FFFF00");
        }
    }

    manager.drawBlobs(blobs);
};
```

4 Physics

4.1 World

The physics engine is based on box2d. The first thing to do if you want to use the physical engine is to initialize the physical world. At each frame, we will step the physics to move the objects to their next position. First, we will create the objects.

```
var ball = new Circle();
var ground = new Rectangle();
```

Then we will setup the physical world and give the objects their physical properties.

```
// Init function
onConnect = function() {
    // Canvas Init
    canvasForeground = $("#canvasForeground").get(0);
    canvasForeground.width=screenW;
    canvasForeground.height=screenH;
    contextForeground = canvasForeground.getContext("2d");

    canvasBackground = $("#canvasBackground").get(0);
```

```

canvasBackground.width=screenW;
canvasBackground.height=screenH;
contextBackground = canvasBackground.getContext("2d");

// Manager Init
manager.setScreenSize(screenW, screenH);
manager.setOnGesture(gesture);
// Set the gravity for the world. 0 on the X axis and 300 on the Y
axis
manager.setGravity(0,300);
// Initialize the world
manager.initPhysics();

// Add a new shape
ball.setCanvasSize(canvasForeground.width, canvasForeground.height);
ball.setContext(contextForeground);
ball.setXRelativePosition(0.5);
ball.setYRelativePosition(0.5);
ball.setRelativeRadius(0.03);
ball.setColor("#FFFF00");
// Set the density of the ball. 1 means that the ball will be fully
affected by the forces
ball.setDensity(1);
// Add the physical body to the ball. The ball will give back 60% of
the forces on impact.
ball.addPhysics(manager.getWorld(), 0.6);
manager.addCircle(ball);

// Add the ground
ground.setCanvasSize(canvasBackground.width,
canvasBackground.height);
ground.setContext(contextBackground);
ground.setXRelativePosition(0.5);
ground.setYRelativePosition(1);
ground.setRelativeWidth(1);
ground.setRelativeHeight(0.1);
ground.setColor("#009FE3");
// Set the density of the ground to 0. It means the ground will not
be affected by gravity since it has no mass.
ground.setDensity(0);
// Add the physics to the ground and set it restitution to 0.6
ground.addPhysics(manager.getWorld(), 0.6);
manager.addRectangle(ground);

// Fill the screen with black
contextBackground.fillStyle = "#000000";
contextBackground.fillRect(0, 0, canvasBackground.width,
canvasBackground.height);
};

```

At last, we will step the physics at each frame.

```

// Draw the objects
draw = function() {

```

```

contextForeground.clearRect(0, 0, canvasForeground.width,
    canvasForeground.height);

// Step the physics and move the objects.
manager.stepPhysics(1.0/30);

manager.drawCircle();
manager.drawRectangle();
};

```

4.2 LinearVelocity and LinearDamping

If you want an object to move by its own, it is possible to add a linear velocity to this object. If you want more realistic behaviour, you can add linear damping. We will see the difference in a race between two balls :

```

var ball1 = new Circle();
var ball2 = new Circle();

```

Both the balls will have the same velocity but one of them will have a damping coefficient

```

// Init function
onConnect = function() {
    // Canvas Init
    .
    .
    .

    // Manager Init
    manager.setScreenSize(screenW, screenH);
    manager.setOnGesture(gesture);
    // No gravity this time
    manager.initPhysics();

    // Add a new shape
    ball1.setCanvasSize(canvasForeground.width, canvasForeground.height);
    ball1.setContext(contextForeground);
    ball1.setXRelativePosition(0.1);
    ball1.setYRelativePosition(0.3);
    ball1.setRelativeRadius(0.03);
    ball1.setColor("#FFFF00");
    ball1.setDensity(1);
    ball1.addPhysics(manager.getWorld(),0.6);
    Sometimes you will want an object to collide with an other
    object but not with all the others. For that you can
    define some collision mask. \\\*
    As always, define your objects :
    \begin{lstlisting}
var ball1 = new Circle();
var ball2 = new Circle();
var ground = new Rectangle();

```

Then set their physical attributes :

```
// Init function
onConnect = function() {
    // Canvas Init
    .
    .
    .

    // Manager Init
    manager.setScreenSize(screenW, screenH);
    manager.setOnGesture(gesture);
    manager.setGravity(0,300);
    manager.initPhysics();

    // Add a new shape
    ball1.setCanvasSize(canvasForeground.width, canvasForeground.height);
    ball1.setContext(contextForeground);
    ball1.setXRelativePosition(0.5);
    ball1.setYRelativePosition(0.3);
    ball1.setRelativeRadius(0.03);
    ball1.setColor("#FFFF00");
    ball1.setDensity(1);
    // Set the collision filter and the collision mask
    ball1.setCollisionFilter(0x0002);
    ball1.setCollisionMask(0x0004);
    ball1.addPhysics(manager.getWorld(),0.6);
    manager.addCircle(ball1);

    ball2.setCanvasSize(canvasForeground.width, canvasForeground.height);
    ball2.setContext(contextForeground);
    ball2.setXRelativePosition(0.5);
    ball2.setYRelativePosition(0.7);
    ball2.setRelativeRadius(0.02);
    ball2.setColor("#00FF00");
    ball2.setDensity(1);
    // Set the same collision filter and the same collision mask than
    // the other ball, so they won't collide
    ball2.setCollisionFilter(0x0002);
    ball2.setCollisionMask(0x0004);
    ball2.addPhysics(manager.getWorld(),0.6);
    manager.addCircle(ball2);

    ground.setCanvasSize(canvasBackground.width,
        canvasBackground.height);
    ground.setContext(contextBackground);
    ground.setXRelativePosition(0.5);
    ground.setYRelativePosition(1);
    ground.setRelativeWidth(1);
    ground.setRelativeHeight(0.1);
    ground.setColor("#009FE3");
    ground.setDensity(0);
    // Set the opposite filter and mask than the balls. The balls will
    // collide with the ground but not with each other.
```

```

ground.setCollisionFilter(0x0004);
ground.setCollisionMask(0x0002);
ground.addPhysics(manager.getWorld(), 0.6);
manager.addRectangle(ground);

// Fill the screen with black
contextBackground.fillStyle = "#000000";
contextBackground.fillRect(0, 0, canvasBackground.width,
    canvasBackground.height);
};

```

And don't forget to draw the objects and to step the physics :

```

// Draw the objects
draw = function() {
    contextForeground.clearRect(0, 0, canvasForeground.width,
        canvasForeground.height);
    manager.stepPhysics(1.0/30);
    manager.drawCircle();
    manager.drawRectangle();
};

```

5 Gesture

5.1 Touch, Tap and release

The gesture recognition is based on the hammer.js framework. We will start with the touch and release event. We will also see the tap event (fast touch and release). All the processing will be in the gesture function. The gesture event are linked to the patterns. Thanks to that, it is possible to make a difference on a gesture event, depending on which object moved. The example will print on the screen the current event : touch or release. At each tap, the printing color will change. We will start by creating a text and a color variable. We will add a default pattern too.

```

var def = new Pattern();
var text = "";
var color = "#FFFFFF";

```

Then we will complete the gesture function.

```

// Gesture recognition function
gesture = function(event) {
    // On release
    if (event.type == "release") {
        text = "Release";
    }
    // On touch
    } else if (event.type == "touch") {
        text = "Touch";
    }
    // On tap
    } else if (event.type == "tap") {

```

```

        if (color != "#FFFFFF") {
            color = "#FFFFFF";
        } else {
            color = "#FF0000";
        }
    }
};

```

Don't forget to enable the gesture recognition for the pattern.

```

// Init function
onConnect = function() {
    // Canvas Init
    .
    .
    .

    // Pattern Init
    def.enableGesture(true);
    manager.addPattern(def);

    // Manager Init
    .
    .
    .
};

```

Finally we will print the text on the screen

```

// Draw the objects
draw = function() {
    contextForeground.clearRect(0, 0, canvasForeground.width,
        canvasForeground.height);

    // Set the font size and color
    contextForeground.font = "30px Arial";
    contextForeground.fillStyle = color;
    // Print the text
    contextForeground.fillText(text, 50, 50);
};

```

5.2 Drag, pinch and rotate

The drag, pinch and rotate events are more complicated to manage because they have an effect on an object. The scale and rotation events don't give absolute values, so it is necessary to register the previous value yourself. We will create a rectangle to demonstrate the effect of the events.

```

var def = new Pattern();
var rect = new Rectangle();
// At first, the angle value is 0
var oldAngle = 0;

```

```
// At first, the scale of the rectangle is 1
var oldScale = 1;
```

We will complete the gesture function.

```
// Gesture recognition function
gesture = function(event) {
    //Get the objects at the gesture location
    var objects = manager.getObjects(event.gesture.touches[0].pageX,
        event.gesture.touches[0].pageY);
    // End of pinch event, rotate event and drag event
    if (event.type == "release") {
        oldScale = 1;
        oldAngle = 0;
    }
    // On pinch
    } else if (event.type == "pinch") {
        // If there is an object at the gesture location
        if (objects[0]) {
            // Rescale the object
            if (event.gesture.scale > oldScale + 0.05 ||
                event.gesture.scale < oldScale - 0.05) {
                objects[0].setScale(objects[0].getScale() +
                    event.gesture.scale - oldScale);
                oldScale = event.gesture.scale;
            }
        }
    }
    // On Rotate
    } else if (event.type == "rotate") {
        // If there is an object at the gesture location and if this
        // object can rotate
        if (objects[0].setAngle) {
            // Rotate the object
            if (event.gesture.rotation > oldAngle + 1 ||
                event.gesture.rotation < oldAngle - 1) {
                objects[0].setAngle(objects[0].getAngle() +
                    event.gesture.rotation - oldAngle);
                oldAngle = event.gesture.rotation;
            }
        }
    }
    // On Drag
    } else if (event.type == "drag") {
        // If there is an object at the gesture location
        if (objects[0]) {
            // Move the object
            objects[0].setXAbsolutePosition(event.gesture.touches[0].pageX);
            objects[0].setYAbsolutePosition(event.gesture.touches[0].pageY);
        }
    }
};
```

Don't forget to activate the gesture recognition for the pattern and to add the rectangle to the scene.

```
// Init function
```

```

onConnect = function() {
  // Canvas Init
  .
  .
  .

  // Pattern Init
  def.enableGesture(true);
  manager.addPattern(def);

  // Manager Init
  manager.setScreenSize(screenW, screenH);
  manager.setOnGesture(gesture);

  // Add a shape
  rect.setCanvasSize(canvasForeground.width, canvasForeground.height);
  rect.setContext(contextForeground);
  rect.setXRelativePosition(0.5);
  rect.setYRelativePosition(0.5);
  rect.setRelativeWidth(0.1);
  rect.setAbsoluteHeight(rect.getAbsoluteWidth());
  rect.setColor("#FFFF00");
  manager.addRectangle(rect);

  // Fill the screen with black
  contextBackground.fillStyle = "#000000";
  contextBackground.fillRect(0, 0, canvasBackground.width,
    canvasBackground.height);
};

```

We now have to draw the rectangle.

```

// Draw the objects
draw = function() {
  contextForeground.clearRect(0, 0, canvasForeground.width,
    canvasForeground.height);

  manager.drawRectangle();
};

```

6 Network

6.1 Server

At first we will see how to create a server and how to send and received simple messages. The network part is based on the socket.io framework.

Servers are written in JavaScript. You can run them using node.js. The server architecture is pretty simple, you just have to define all type of messages that can be exchanged and then describe what to do with those messages.

```

var app = require('http').createServer(handler),
    io = require('socket.io').listen(app),

```



```

fs = require('fs')

app.listen(1337);

function handler (req, res) {
  fs.readFile(__dirname + '/index.html',
    function (err, data) {
      if (err) {
        res.writeHead(500);
        return res.end('Error loading index.html');
      }

      res.writeHead(200);
      res.end(data);
    });
}

io.sockets.on('connection', function (socket) {
  // When a client is connected

  // Send a welcome message to the new client
  socket.emit('hello', "Hello World !");

  // When a client sends a message of the 'color' type, broadcast
  // the message to the other clients
  socket.on('color', function (data) {
    socket.broadcast.emit('color', data);
  });

  socket.on('disconnect', function() {
    // When a client is disconnected
  });
});

```

When receiving a message, the server can broadcast it or send it to a specific client. To send a message to a specific client :

```
io.sockets.socket(idOfTheClient).emit('type', data);
```

The id of a client can be found with :

```

idOfTheClient = socket.id;
\end{lstlisting}
When the server is ready, you have to create the client. Don't
forget to change the ip address or port if it's necessary.
\begin{lstlisting}
var def = new Pattern();
var color = "#FFFFFF";
var socket = io.connect("127.0.0.1:1337");

```

As usual, add the pattern to the manager. `/**` Then complete the draw function.

```

// Draw the objects
draw = function () {

```

```

contextForeground.clearRect(0, 0, canvasForeground.width,
    canvasForeground.height);

var blobs = client.getTuioblobs();

// Each time you touch the screen, send a message to the server to
// change the color of the text
for (var i in blobs) {
    socket.emit('color', "");
}

// Print the text "Network"
contextForeground.font = "30px Arial";
contextForeground.fillStyle = color;
contextForeground.fillText("Network", 50, 50);
};

```

The last thing to do is to define what to do when receiving a message.

```

// When receiving a "hello" message
socket.on('hello', function (data) {
    // Print the text in green
    color = "#00FF00";
});

// When receiving a "color" message
socket.on('color', function (data) {
    // Change the text color
    if (color != "#FFFFFF") {
        color = "#FFFFFF";
    } else {
        color = "#FF0000";
    }
});

```

6.2 Messages and objects

In this section we will see how we can attach data to a message. We will send the position of the blobs between the clients, and then draw them.

```

var def = new Pattern();
// Store the blobs received by messages.
var networkBlobs = [];
var socket = io.connect("127.0.0.1:1337");

```

Add the default pattern to the manager, then complete the drawing function

```

// Draw the objects
draw = function () {
    contextForeground.clearRect(0, 0, canvasForeground.width,
        canvasForeground.height);

    var blobs = client.getTuioblobs();

```

```

// For each blob, send the position and the size to the server
for (var i in blobs) {
    msg = {xPos: blobs[i].getX(), yPos:blobs[i].getY(),
          width:blobs[i].getWidth(), height: blobs[i].getHeight()};
    socket.emit('blob', msg);
}

// Draw each received blob on the screen
for (var i = 0; i < networkBlobs.length; i++) {
    def.draw(networkBlobs[i]);
}

// Clear the array
networkBlobs = [];
};

```

Create a new blob when the server send you a message. When receiving an object, you need to deserialized the message before using it.

```

// When receiving a "blob" message
socket.on('blob', function (data) {
    // Deserialize the message in an array of objects
    var obj = JSON.parse(data);
    // Create a new blob with the information received
    blob = new Tuio.Blob({
        si : 1,
        bi: -1,
        xp: obj[0].xPos,
        yp: obj[0].yPos,
        a: 0,
        xs: 0,
        ys: 0,
        rs: 0,
        ma: 0,
        ra: 0,
        w: obj[0].width,
        h: obj[0].height,
        ar: obj[0].width * obj[0].height
    });

    // Add a context to the blob
    blob.setContext(contextForeground);

    // Add the new blob to the array
    networkBlobs.push(blob);
});

```

7 Phones and Tablets

7.1 Touch Events

Since tablets don't give you TUIO events, you have to find an other way to get the touch events. But you don't want to lose the blob objects that is an important part of the manager. What we will do is to get the touch events from the browser and create blobs at the right places. Some information will be lost, as the angle or the speed. Be careful to use a browser that accept touch events.

The new minimal code for phones and tablets is :

```
$.noConflict();
jQuery(document).ready(function($) {
    Hammer.plugins.fakeMultitouch();
    var screenW = $(window).width();
    var screenH = $(window).height();
    var manager = new Manager();
    var canvasForeground = null;
    var contextForeground = null;
    var canvasBackground = null;
    var contextBackground = null;
    var def = new Pattern();
    var onGoingTouches = [];

    // Gesture Recognition Function
    gesture = function(event) {
        if (event.type == "release") {
            // End of pinch event, rotate event and drag event
        }
    };

    // Canvas Init
    canvasForeground = $("#canvasForeground").get(0);
    canvasForeground.width = $(window).width();
    canvasForeground.height = $(window).height();
    contextForeground = canvasForeground.getContext("2d");

    canvasBackground = $("#canvasBackground").get(0);
    canvasBackground.width = $(window).width();
    canvasBackground.height = $(window).height();
    contextBackground = canvasBackground.getContext("2d");

    // Manager Init
    manager.setScreenSize($(window).width(), $(window).height());
    manager.setOnGesture(gesture);
    manager.addPattern(def);

    // Fill the screen with black
    contextBackground.fillStyle = "#000000";
    contextBackground.fillRect(0,0,canvasBackground.width,canvasBackground.height);

    // Add a "touch start" event listener.
    canvasForeground.addEventListener('touchstart', function(event) {
```

```

event.preventDefault();
var touches = event.changedTouches;

// When a touch starts, create a blob and add it to the
// OnGoingTouches array.
for (var i = 0; i < touches.length; i++) {
    var idx = -1;
    for (var j = 0; j < onGoingTouches.length; j++) {
        if (onGoingTouches[j].getSessionId() ==
            touches[i].identifier) {
            idx = j;
        }
    }
    if (idx == -1) {
        var blob = new Tuio.Blob({
            si: touches[i].identifier,
            bi: -1,
            xp: touches[i].pageX / $(window).width(),
            yp: touches[i].pageY / $(window).height(),
            a: 0,
            xs: 0,
            ys: 0,
            rs: 0,
            ma: 0,
            ra: 0,
            w: touches[i].webkitRadiusX / $(window).width(),
            h: touches[i].webkitRadiusY / $(window).height(),
            ar: touches[i].webkitRadiusX / $(window).width() *
                touches[i].webkitRadiusY / $(window).height()
        });
        onGoingTouches.push(blob);
    }
}
},false);

// Add a "touch move" event listener
canvasForeground.addEventListener('touchmove', function(event) {
    event.preventDefault();
    var touches = event.changedTouches;
    // Update the blob in the OnGoingTouches array
    for (var i = 0; i < touches.length; i++) {
        for (var j = 0; j < onGoingTouches.length; j++) {
            if (onGoingTouches[j].getSessionId() ==
                touches[i].identifier) {
                var blob = new Tuio.Blob({
                    si: touches[i].identifier,
                    bi: -1,
                    xp: touches[i].pageX / $(window).width(),
                    yp: touches[i].pageY / $(window).height(),
                    a: 0,
                    xs: 0,
                    ys: 0,
                    rs: 0,
                    ma: 0,

```

```

        ra: 0,
        w: touches[i].webkitRadiusX / $(window).width(),
        h: touches[i].webkitRadiusY / $(window).height(),
        ar: touches[i].webkitRadiusX / $(window).width() *
            touches[i].webkitRadiusY / $(window).height()
    });
    onGoingTouches.splice(j, 1, blob);
    }
    }
    }, false);

// Add a "touch end" event listener
canvasForeground.addEventListener('touchend', function(event) {
    event.preventDefault();
    var touches = event.changedTouches;
    // Remove the blob from the OnGoindTouches array
    for (var i = 0; i < touches.length; i++) {
        for (var j = 0; j < onGoingTouches.length; j++) {
            if (onGoingTouches[j].getSessionId() ==
                touches[i].identifier) {
                onGoingTouches.splice(j,1);
            }
        }
    }
    }, false);

// Add a "touch cancel" event listener
canvasForeground.addEventListener('touchcancel', function(event) {
    event.preventDefault();
    var touches = event.changedTouches;
    // Remove the blob from the OnGoindTouches array
    for (var i = 0; i < touches.length; i++) {
        for (var j = 0; j < onGoingTouches.length; j++) {
            if (onGoingTouches[j].getSessionId() ==
                touches[i].identifier) {
                onGoingTouches.splice(j,1);
            }
        }
    }
    }, false);

// Add a "touche leave" event listener
canvasForeground.addEventListener('touchleave', function(event) {
    event.preventDefault();
    var touches = event.changedTouches;
    // Remove the blob from the OnGoindTouches array
    for (var i = 0; i < touches.length; i++) {
        for (var j = 0; j < onGoingTouches.length; j++) {
            if (onGoingTouches[j].getSessionId() ==
                touches[i].identifier) {
                onGoingTouches.splice(j,1);
            }
        }
    }
    }, false);

```

```

    }
  },false);

  // Draw the objects
  setInterval(draw = function() {
    contextForeground.clearRect(0,0,canvasForeground.width,
      canvasForeground.height);

    // Draw the blobs
    for (var i = 0; i < onGoingTouches.length; i++) {
      if (manager.blobMatchPattern(onGoingTouches[i],def)) {
        onGoingTouches[i].setContext(contextForeground);
        def.draw(onGoingTouches[i]);
      }
    }

    // Draw the objects
    manager.drawRectangle();
    manager.drawCircle();
  },33);
});

```

7.2 Sensors

Phones and tablets always have a bunch of sensors that you can use if your browser allows you do to it. Gyroscopes works pretty well most of the time but accelerometer aren't reliable. In this example we will move a ball when tilting the device.

Start by creating a circle and the angles for the two axis we are interested in.

```

var circle = new Circle();
var beta = 0;
var gamma = 0;

```

Give the ball its attribute :

```

// Add a ball
circle.setCanvasSize(canvasForeground.width, canvasForeground.height);
circle.setContext(contextForeground);
circle.setRelativeRadius(0.03);
circle.setXRelativePosition(0.5);
circle.setYRelativePosition(0.5);
manager.addCircle(circle);

```

Define a new listener for the gyroscope :

```

// If the browser supports the device orientation events
if (window.DeviceOrientationEvent) {
  // Add an event listener and store the angle values.
  window.addEventListener("deviceorientation", function(event) {

```

```

        beta = event.beta;
        gamma = event.gamma;
    }, false);
}

```

Finally, move the ball :

```

// Draw the objects
setInterval(draw = function() {
    contextForeground.clearRect(0,0,canvasForeground.width,
        canvasForeground.height);

    // Move the ball in function of the angle values
    if (beta > 10) {
        circle.setYAbsolutePosition(circle.getYAbsolutePosition() + 5);
    } else if (beta < -10) {
        circle.setYAbsolutePosition(circle.getYAbsolutePosition() - 5);
    }

    if (gamma > 10) {
        circle.setXAbsolutePosition(circle.getXAbsolutePosition() + 5);
    } else if (gamma < -10) {
        circle.setXAbsolutePosition(circle.getXAbsolutePosition() - 5);
    }

    // Draw the objects
    manager.drawRectangle();
    manager.drawCircle();
},33);

```

8 Optimization

8.1 FPS

A good way to decrease the load on the CPU is to control the number of frames per second. The draw function is called each time the page received a TUIO message. You can change that by calling the function only each X milliseconds.

```

setInterval(draw = function() {
    //Draw function
},33);

```

Two important points : `/**` - If you decide to call the function draw at regular intervals, you have to remove the `onRefresh` function. Otherwise, you will overload your CPU more than before.

- The number at the end of the `setInterval` function (here 33) is not the number of frames per second but the time between two frames (33ms means 30fps).

When you run some multiscreen applications, if you want the objects to move at the same speed on each of the devices, you need to set the same fps for

everyone.

8.2 Memory Management

A very important point if you want your application to run smoothly is to delete the objects when you no longer need them. The JavaScript garbage collector is supposed to do that for you but you will have much better performances if you do it yourself. Always make sure that if an object isn't supposed to be used anymore (out of the screen for example), it is deleted.

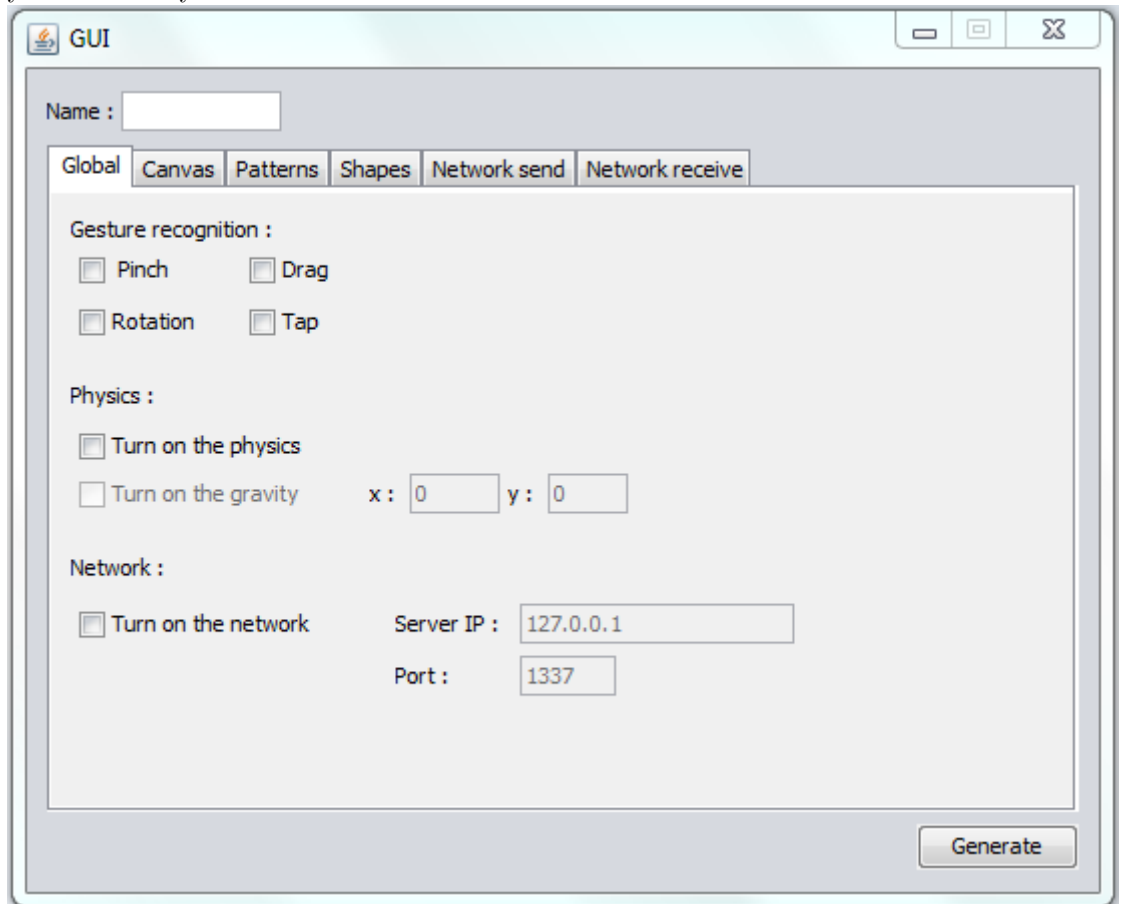
When using the physical engine, you will need to remove the physics of the shape before deleting the shape, otherwise, the physical body of the shape will remain in the scene. If you don't delete the physical bodies of your shapes, an error of the physical engine may occur.

8.3 Network

If you don't want to overload the network, try to send only the information that you need. Don't send all the properties of an object if you only need some of them. Serialization and deserialization are expensive. Don't send the information in real time if you don't need them. If you need the data only 1 time per second so send the messages only 1 time per second.

9 GUI

The framework also have a graphical user interface to help you to generate your code easily.

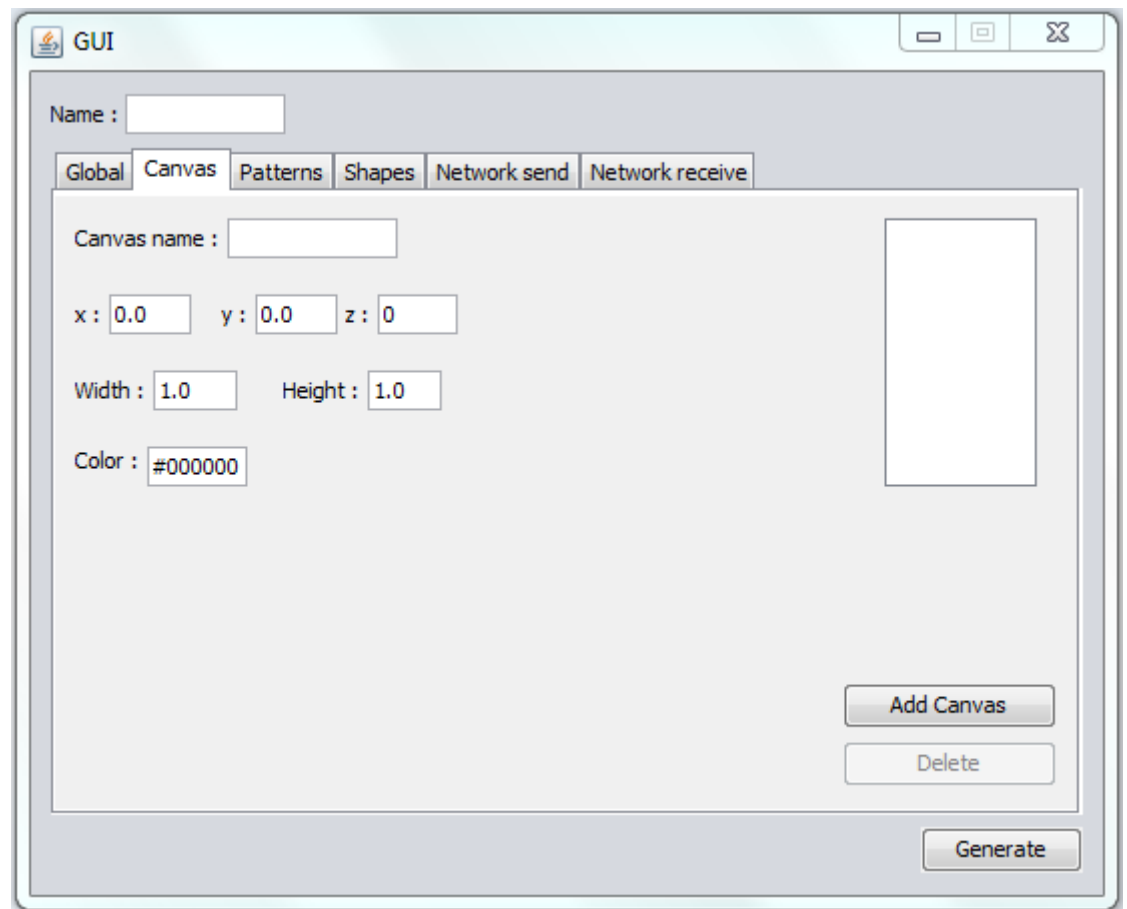


The screenshot shows a graphical user interface window titled "GUI". At the top, there is a "Name :" label followed by an empty text input field. Below this is a tabbed interface with six tabs: "Global", "Canvas", "Patterns", "Shapes", "Network send", and "Network receive". The "Global" tab is currently selected. Inside the "Global" tab, there are three sections of settings:

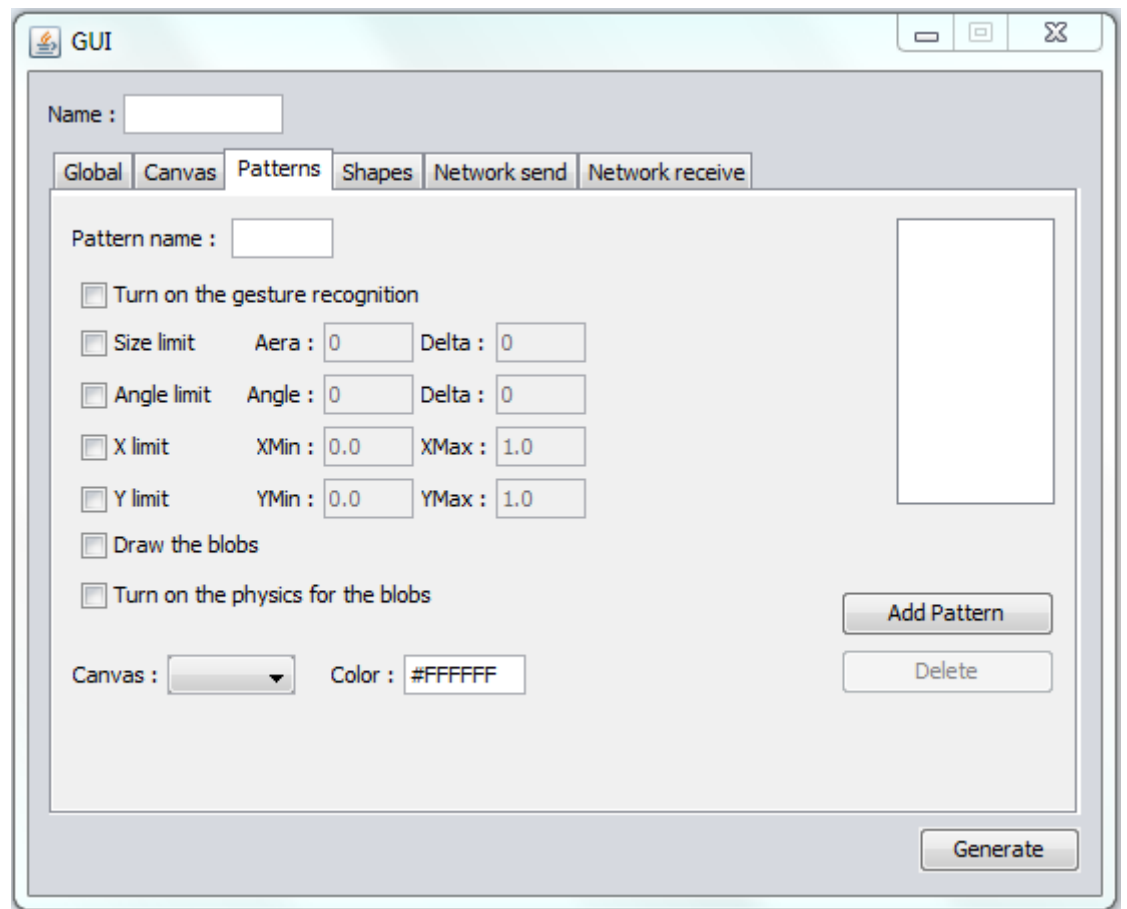
- Gesture recognition :** This section contains four checkboxes: "Pinch", "Drag", "Rotation", and "Tap". All four checkboxes are currently unchecked.
- Physics :** This section contains two checkboxes: "Turn on the physics" and "Turn on the gravity". Both are unchecked. To the right of these checkboxes are two text input fields labeled "x :" and "y :", both containing the value "0".
- Network :** This section contains one checkbox, "Turn on the network", which is unchecked. To its right are two text input fields: "Server IP :" containing "127.0.0.1" and "Port :" containing "1337".

At the bottom right of the window, there is a "Generate" button.

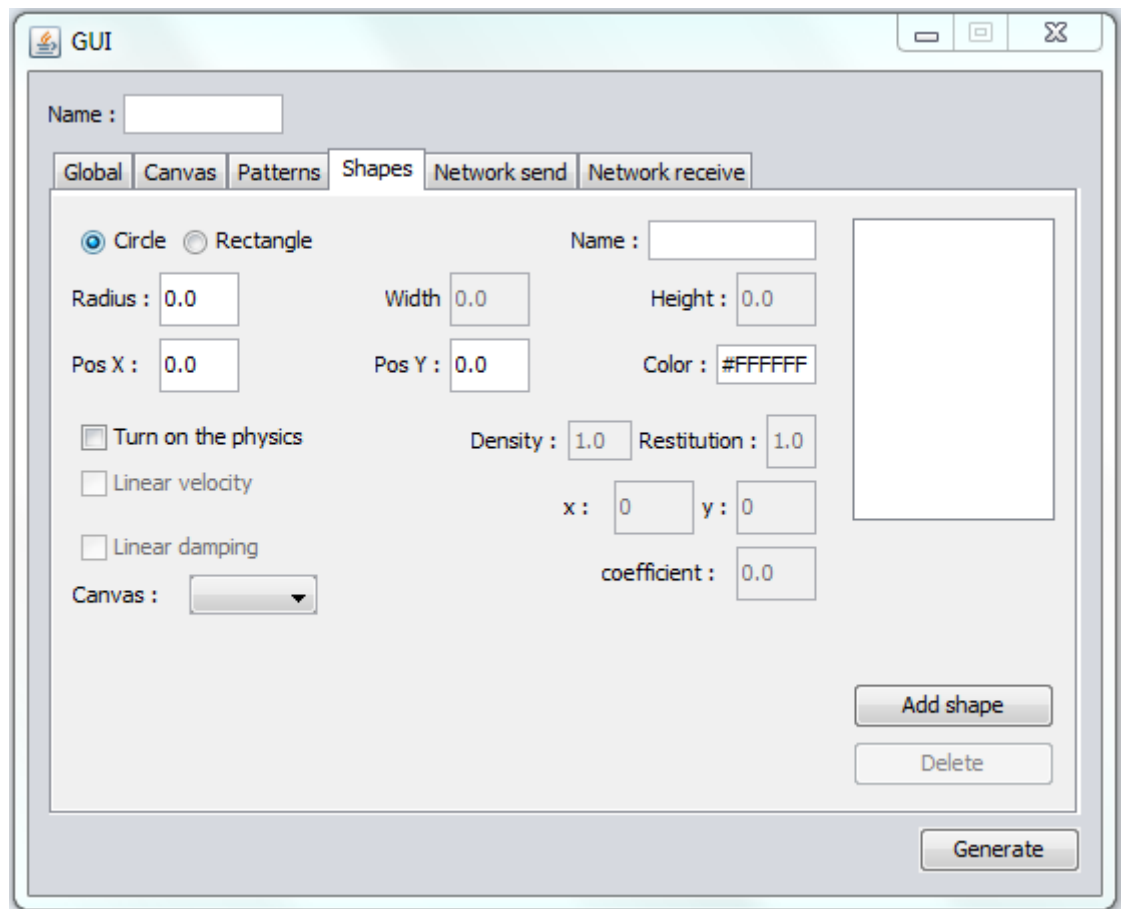
On this window you can choose if you want to turn on or off the gesture recognition, the physics and the network for your application.



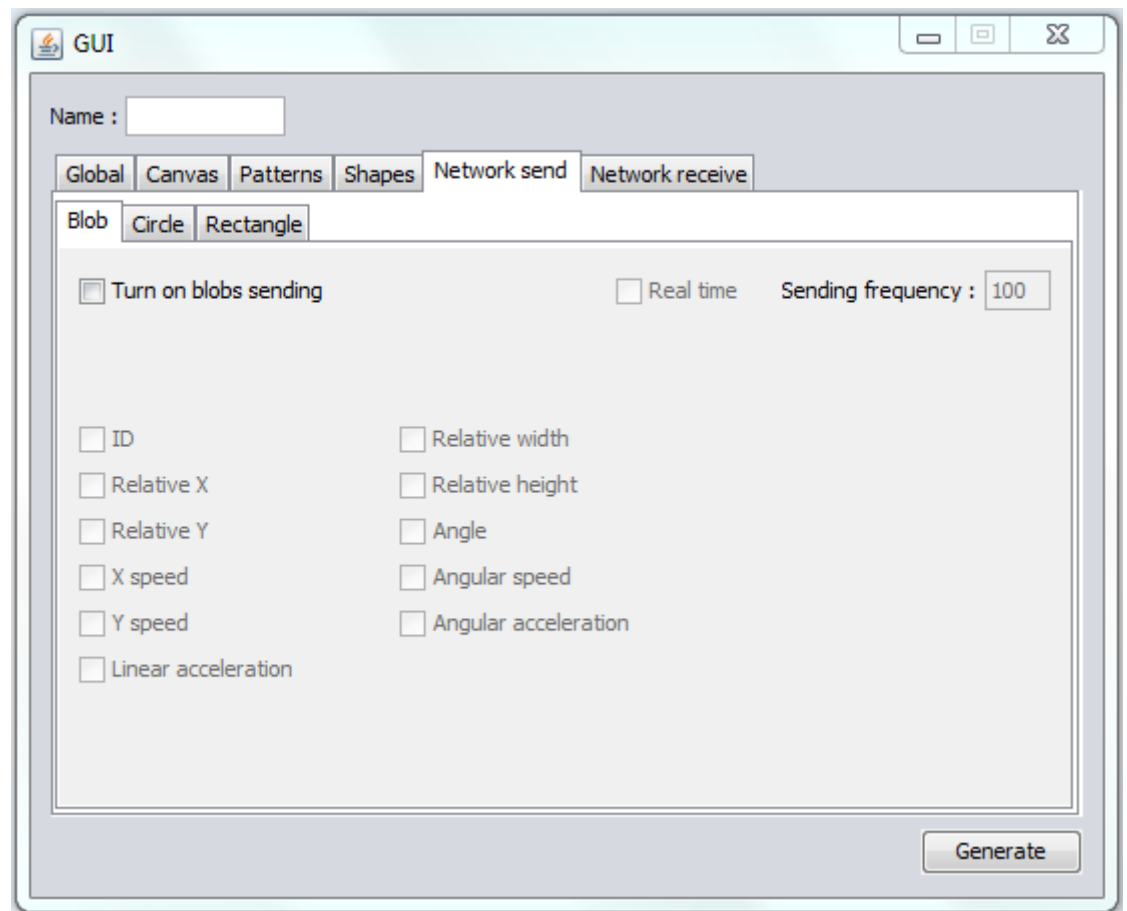
The canvas window allows you to define and add canvas to your application. The X, Y, width and height are relative to the screen size. Z is the layer of the canvas.



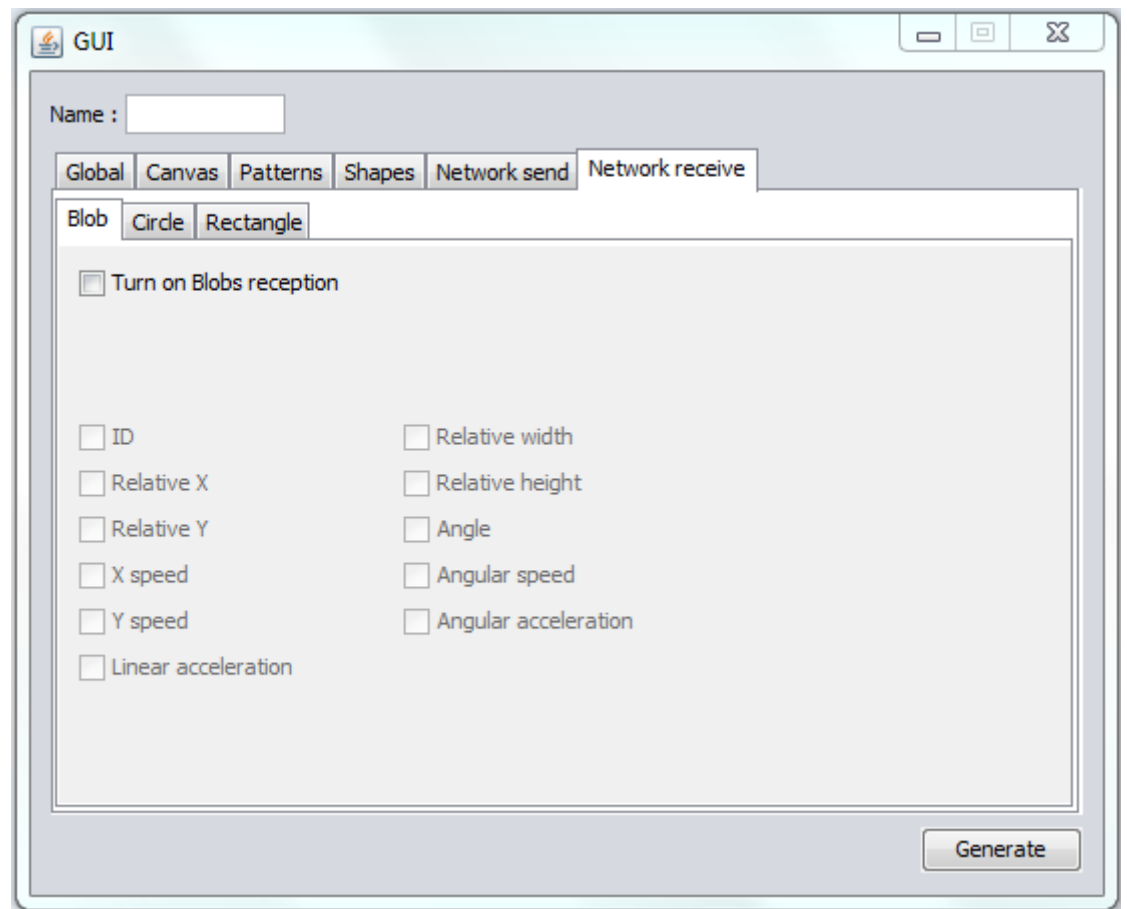
The Pattern window allows you to define and add patterns to your application.
If you choose a x or a y limit, the numbers are relative to the screen size.



The Shape window allows you to define and add shapes to your application. Positions, width, height and radius are relative to the screen size.



You can choose witch properties of the objects you want to send on the network with that window. You can also set if you want the blobs to be sent in real time or only once in a while.



The last window allows you to receive objects through the network. You can choose which properties you will receive.