SmartFoxServer 2X™
massive multiplayer platform

SFS2X Cannon Combat

Mobile Multi-Player Physics Game

DOCUMENTATION / WALK-THROUGH

Release Date: March 2012

Written By: Fabricio Medeiros & Wayne Helman, *A51 Integrated*

a51

Cannon Combat Walk-Through

Introduction

The Cannon Combat game is a sample SFS2X application that utilizes the Adobe AIR 2 runtime and deploys the same code base to multiple mobile platforms. The game itself is based on the popular Angry Birds game developed by Rovio Mobile. It is a turn based game where a cannon shoots a ball at their opponents desk items. The physics is achieved with the use of the Box2D engine and the game can be deployed on both Apple iOS devices and Android devices.

Setting Up The SmartFoxServer Zone

We're going to assume you have a running SmartFoxServer 2X already installed either locally or on a remote server. Perform the following steps to install the included zone:

1. Copy the CannonCombat.zone.xml file to your server's /SFS2X/zones directory.
2. Copy the /cannonCombat folder with the extension JAR file to your server's /SFS2X/extensions directory.
3. Restart the server.
4. Launch the SFS2X Administration Tool to verify that the zone has been successfully loaded.

Setting Up The Project

The Cannon Combat game is built with two distinct projects that use the same code base. The first version (dev) runs in AIR 2 for testing purposes and will run on your workstation. The second version (prod) runs on the specified mobile device supported by Flash Builder 4.5.

In order to share the same common source package, both projects have an external source folder added in *Project -> Properties -> Flex Build Path -> Source Path*

Therefore, they differentiate in their Default Application mxml file only and obviously, descriptors.

This project requires the Flex SDK 4.6, otherwise, you will need to add two SWC files (mobilecomponents.swc and mobile.swc) manually through *Project -> Properties -> Flex Build Path -> Library Path* for the dev version only. These two SWC files can be found in the Flex SDK 4.6.

If after including both SWC files as described above, the theme "Mobile" doesn't appear under the option *Flex Theme* in the project settings, please add it manually.

For the production version, you will also need a .p12 certificate and a .mobileprovision file in order to deploy to iOS devices (this is not required for Android). Please see the following link for detailed instructions on how to provision for iOS and the App Store:

http://www.adobe.com/devnet/air/articles/packaging-air-apps-ios.html

Since we use SDK 4.6, the descriptor's application tag uses version 3.1, as below:

```
<application xmlns="http://ns.adobe.com/air/application/3.1">
```

Setting Up Eclipse For Java Extensions

1. *File -> New -> Java Project*
2. Type a Name for the project like "SFS2XCannonCombatExt"
3. Select Create project from existing source, and navigate to the extracted Java source code included in the download.
4. Click on Finish to import the project and follow the prompts.

Tutorial Conventions

Keeping code style consistent is an important factor when working with a team and even more important when teaching and sharing. In our sample code we use some conventions that should be defined prior to delving into the ActionScript code.

Class properties always begin with an underscore e.g.

```
private var _someVar:String = 'foobar';
```

Method arguments always begin with the dollar symbol e.g.

```
private function handleLogin($event:MouseEvent):void
```

Local variable have no prefix e.g.

```
var username:String = 'mr.foo';
```

Multiple declarations are separated by a comma e.g.

```
var    username      :String = 'mr.foo',
       password      :String = 'razzamataz',
       userId        :int = 244;
```

Limitations Developing Multi-User Apps with Adobe AIR

Developing multi-user application for AIR poses one major problem that Flash applications do not experience. For development and testing, Flash is capable of running multiple instances of an application on a single system. With AIR, only a single instance of an application can be run on a single machine. You cannot even run one installed application and try to run the same application in the FlashBuilder IDE. This limitation makes it very difficult to develop multi-user applications since testing your application requires two physically separate computers or a host running a virtualized second instance of an operating system.

Because of this limitation, we've developed a system within a single AIR application that makes use of the `NativeWindow()` class to instantiate multiple instances of our game in a single AIR application. There are

however issues that arise using this method that are beyond the scope of this document.

Code Structure

There are 3 views (User List, Game List and Chat) in the lobby, use the bar at the bottom of the views to navigate between them.

- The Game List view provides the facility to join a game as a Spectator.
- The Chat view is just a simple public chat system.
- The User List allows players to challenge other players in a 2-player game.

After tapping on challenge, one has to select an arena (only a single arena is included in the source code – Arena01) and the invitee will receive a flashing notification on the bottom left corner of the view.

This mechanism relies on the invitation features provided by SFS2X.

LobbyController, line 838:

```
// Send the invitation; recipients have 10 seconds to reply before the invitation expires
var update:ISFSObject = new SFSObject();
update.putInt('arena',$arena);

_sfs.send(new InviteUsersRequest([_gameRoom.invitee], 10, update));

obj.invitee = _gameRoom.invitee;

_winChallenge = new WinChallenge(obj, _sfs);
addChild(_winChallenge);
```

If the invitee denies the challenge, we send an `InvitationReplyRequest` back to the server with a refuse message.

To avoid creating unnecessary rooms, the invitee is responsible for creating the room once the invite is accepted.

Upon invite acceptance, we send an accept message and dispatch a custom event to the Lobby Controller. Subsequently, we move forward to the arena and create the game room, where the room name is as seen on line 597 of the LobbyController:

```
settings.name = $inviter.name +'_vs_'+$invitee.name;
```

The events and requests to handle this transaction can be found in the ChallengeListItem class.

ChallengeListItem line 111:

```
private function onClick($e:MouseEvent = null):void
{
	var target:Object = $e.currentTarget;

	_soundManager.playSound('pop');

	if(target.name == 'deny')
	{
```

```
                _sfs.send(new InvitationReplyRequest(_data.invitation, InvitationReply.REFUSE));
        }
        else if(target.name == 'accept')
        {
                var objSFS:ISFSObject = new SFSObject as ISFSObject;

                //Save the game room name to be sent to the inviter
                objSFS.putUtfString('room_name', _data.invitation.inviter.name +'_vs_'+_data.
invitation.invitee.name);

                //Propagate event to the lobby to create the room (invitee is the owner)
                dispatchEvent(new CustomEvent(CustomEvent.CHALLENGE_ROOM, true, false, _data.
invitation.inviter, _data.invitation.invitee));

                //Send invitation reply back to the inviter
                _sfs.send(new InvitationReplyRequest(_data.invitation, InvitationReply.ACCEPT,
objSFS));
        }
        destroy();
}
```

We listen for our custom event in the LobbyController class.

LobbyController line 856:

```
private function onChallengeRoom($e:CustomEvent = null):void
{
        //Invitee creates the room if invitation was accepted
        addRoom($e.arg[0], $e.arg[1]);
}
```

NOTE: Invitation requests also have a certain amount of time to expire (we have set it to 10 seconds).

If a user joins the same game as a Spectator, he will only follow the other players turns, never taking one, which means never being able to control the cannon nor set room variables besides `Settings.RV_ARENA` to allow other users to be aware he is in the room and ready to watch. On the other hand, once both actual players hit ready, the room is not “joinable” anymore.

Upon game room creation, a random turn variable is created to determine which player starts.Once a player has been assigned a turn, he/she is able to control their cannon.

On line 310 of Cannon.as we do the necessary calculation and collect the information required:

```
/**
* Collects coordinates data while moving the cannon
*
* @param $e :MouseEvent - MOUSE_MOVE event
*/
private function cannonBallMoved($e:MouseEvent):void
{
        _posX = _cannon.mouseX;
        _posY = _cannon.mouseY;

        //Limit the x axis
        if(_posX < -120)    _posX = -120;
        else if(_posX > 0)  _posX = 0;
```

```
        //Limit the y axis
        if(_posY < -60)     _posY = -60;
        else if(_posY > 60) _posY = 60;

        //Get the distance between those two points to define power
        _p1 = new Point();
        _p2 = new Point(_posX * .5, _posY * .5);
        _distance = Point.distance(_p1, _p2);

        //Define the angle
        _angle = Math.atan2(_posY, _posX * .5) * (180/Math.PI) + 180;

        //Limit the angle (half circle)
        if(_angle >= 90 && _angle <= 180)
        {
              _angle = 90;
        }
        else if(_angle <= 270 && _angle >= 180)
        {
              _angle = 270;
        }

        //Powermeter
        _powerMeter.visible = true;

        _power =  (_distance / 10) + 1;

        //Move the powermeter and scale it according to the intensity
        _powerMeter.x = _cannon.mouseX;
        _powerMeter.y = _cannon.mouseY;
        _powerMeter.scaleX = _powerMeter.scaleY = _power;

        //Rotate the cannon
        _cannon.chamber.rotation = _angle;

        //Broadcast cannon rotation and power to opponent
        var update:ISFSObject = new SFSObject();
        update.putFloat(Settings.OM_CANNON_ANGLE, _angle);

        if(update.size() > 0)      _gameController.broadcastMessage(update);
        //In production this could take advantage of UDP as an Extension instead
}
```

While we move the cannon, we will broadcast an object `OM_CANNON_ANGLE` to update its movement on the opponent's screen.

When the player "taps up" or let the mouse go, we create the physics body on our screen and broadcast a room variable (`Settings.RV_CANNON`) with the necessary properties to the everyone else in the room, so that they can reproduce the same "shot".

Cannon.as line 392:

```
//Box2D physics
var     sphereShape   :b2CircleShape = new b2CircleShape(15/_world.worldScale),
        sphereFixture :b2FixtureDef = new b2FixtureDef(),
        sphereBodyDef :b2BodyDef = new b2BodyDef();
```

```
	sphereFixture.density = 100;
	sphereFixture.friction = .6;
	sphereFixture.restitution = .05;
	sphereFixture.shape = sphereShape;
	sphereBodyDef.type = b2Body.b2_dynamicBody;
	sphereBodyDef.userData = {assetName:'CANNONBALL', assetSprite:_ball, remove:false,
init:false};

	//We determine the fake ball (visual ball asset inside the chamber in the .fla) location in
its grand parent (the whole cannon, not only the chamber)
	_ballPoint = new Point(_fakeBall.x, 0);
	_ballPoint = localToGlobal(_ballPoint);
	_ballPoint = _cannon.chamber.globalToLocal(_ballPoint);

	//On the right hand side we switch direction
	if (_side == 'right')
	{
		_posX *= -1;
		_posY *= -1;

		if(_ballPoint.x > 0)
			_ballPoint.x *= -1;
	}
	else
	{
		_ballPoint.y *= -1;
	}

	//For shooting
	_angle = Math.atan2(_posY, Math.abs(_posX));

	//Negative value (for _posX or _posY) gives you the opposite quadrant, where the actual ball
is
	//We also need to add the cannon position and divide all that by the world scale ratio
	sphereBodyDef.position.Set( (_world.activeCannon.x + _ballPoint.x) / _world.worldScale , (_
world.activeCannon.y + _ballPoint.y) / _world.worldScale);
	sphereBodyDef.linearDamping = .4;

	_sphere = _world.worldB2.CreateBody(sphereBodyDef);
	_sphere.SetBullet(true);
	_sphere.CreateFixture(sphereFixture);

	//On the right hand side direction is inverted (negative) for x only
	//We cut the x component in half cos the screen is small
	(_side == 'right')
	?
		_sphere.SetLinearVelocity(new b2Vec2(-_distance * Math.cos(_angle) * .5, -_distance *
Math.sin(_angle)))
	:
		_sphere.SetLinearVelocity(new b2Vec2(_distance * Math.cos(_angle) * .5, -_distance *
Math.sin(_angle)));

	//Update the room variable with cannonball's angle and distance to be picked up by opponent
in the same room
	var update:ISFSObject = new SFSObject();

	update.putDouble('ballPointX', _ballPoint.x);
	update.putDouble('ballPointY', _ballPoint.y);
	update.putDouble('dist', _distance);
	update.putDouble('angle', _angle);
```

```
_gameController.broadcastRoomVariable(update, Settings.RV_CANNON);
```

On the GameController.as line 449 we are listening for the room variables updates:

```
private function onRoomVarUpdate($e:SFSEvent):void
{
	var	room	:Room			= $e.params.room,
		vars	:Array			= $e.params.changedVars,
		slot	:String			= vars[0],
		obj	:ISFSObject,
		count	:int			= vars.length;

	if(vars.indexOf(Settings.RV_CANNON) != -1)
	{
		obj = room.getVariable(slot).getSFSObjectValue();

		var sender :String = obj.getUtfString('sender');

		if(sender != _myName)
		{
			//Simulate opponent's cannonball on my screen
			_world.activeCannon.oppShoot(obj.getDouble('posX'), obj.getDouble('ballPointX'),
obj.getDouble('ballPointY'), obj.getDouble('dist'), obj.getDouble('angle'));
		}
	}
	else if (vars.indexOf(Settings.RV_TURN) != -1)
	{
		_playerTurn = room.getVariable(slot).getStringValue();

		_world.playerTurn = _playerTurn; //Switch turn int the world logic

		_world.stop();
		_world.reset();

		indicateTurn();
	}
}
```

A timer to shoot is set every time turns switch between the two players, by default it is set to 15 seconds. But the tricky part is after the shot is taken. We need to decide how or when to switch turns and make certain the the physics engine has stopped moving objects an everyone's screen – this is to make sure that everyone in the game room is receiving the exact same experience.

An `ENTER_FRAME` event runs to take care of the physics engine and animations. We add another timer and also rely on checking all physics bodies.

```
TweenLite.delayedCall(Settings.DEFAULT_CLOCK_TIME, onWorldTimer);
```

Adobe recommends to avoid using timers on mobile devices:
http://help.adobe.com/en_US/as3/mobile/flashplatform_optimizing_content.pdf

We only use the timer class on the shooting clock because none of the heavy processing is happening yet. From that point on we use TweenLite's delayed calls.

Initially, when bodies are added to the stage, they are stored in an array which we loop through to check each body's state ("awakeness") and X/Y velocities.

BasicArena, line 523:

```
public function sleepCheck():void
{
	TweenLite.delayedCall(1, sleepCheck);

	var	awake	:Boolean,
		body	:b2Body,
		data	:*;

	_worldAsleep = true;

	for each (body in _arrBodies)
	{
		awake = body.IsAwake();

		if(awake || Math.abs(body.GetLinearVelocity().x) > 0.001 || Math.abs(body.
GetLinearVelocity().y) > 0.001)
		{
			_worldAsleep = false;
			break;
		}
		else
		{
			_worldAsleep = true;
		}
	}

	if(_worldAsleep && _collisionInit)
	{
		_collisionInit = false;

		TweenLite.killDelayedCallsTo(sleepCheck);

		//Tell the controller to not trigger the time expired event
		_gameController.stopWorldTimer();
	}
}
```

Therefore, we switch turns once all bodies are not moving and the timer behind the scenes has expired.
If the bodies reach a sleep state prior to expiring the timer, we favour that scenario and perform a turn switch.

SFS2X Usage Summary

Room Variables:

- To reproduce the logic for a shot on the opponent screen. The room var contains the x, y, distance and angle of the cannonball.
- To control turns.
- To store which arena has been selected by the challenger.

Global Room Variables:

- For game start notification. Client side can only listen for a Reserved Global Room Var.

Extensions:

- For game start notification. Client side fires the extension, so that others can pick it up.

Object Messages:

- For sleep state of bodies.
- For cannon rotation, while one is taking a shot.
- For game over notification.

Public Message:

- Public chat.

Private Message:

- To remove the challenge from the notification list on the invitee screen.

Conclusion

Cannon Combat includes many more gaming concepts that are beyond the scope of this document. The goal here, was to illustrate the ease in which a mobile game can be created in a real world scenario.

There is still more that could be added to this game. For example, Arena selection and leveling could be added. High scores could be kept in a database with player information.

What this game illustrates is the power, flexibility, and possibilities of the SmartFoxServer 2X framework on mobile devices. With the many features, APIs, expandability, and built in functionality available to developers, SmartFoxServer 2X is a superb choice for multi-player casual and hard core gaming environments and applications.

Notice

Contact gotoAndPlay() if you are interested in buying a commercial license which removes the above restrictions.

Contributors

Produced By:	A51 Integrated, Toronto, Ontario, http://a51integrated.com
ActionScript Development:	Fabricio Medeiros
Java Development:	Andy Rohan
Graphic Design:	Steve Schoger
Project Management:	Sue Timmins
Testing & QA:	Fabricio Mederios, Sue Timmins
Documentation:	Fabricio Medeiros, Wayne Helman