Evaluation Task

Heuristic Evaluation

Heuristics for Playability

The following heuristics have been suggested in the CHI conferences papers "Using Heuristics to Evaluate the Playability of Games" (2004) and "Heuristic Evaluation for Games: Usability Principles for Video Game Design" (2008).

Initial Experience

Upon initially turning the game on the player has enough information to get started to play.
"USING HEURISTICS TO EVALUATE THE PLAYABILITY OF GAMES" (2004)

As pinball is a simple and well-known game, the player should be allowed to directly start playing the game without having to read any instructions or a manual.

Function of Visual Elements is self-explanatory

Art should be recognizable to player, and speak to its function.
"USING HEURISTICS TO EVALUATE THE PLAYABILITY OF GAMES" (2004)

The function of all the game elements like bumpers, flippers and the gaming field itself should be obvious even to a first-time user.

Provide users with information on game status

Users make decisions based on their knowledge of the current status of the game. Users should be provided with enough information to allow them to make proper decisions while playing the game.
"HEURISTIC EVALUATION FOR GAMES: USABILITY PRINCIPLES FOR VIDEO GAME DESIGN" (2008)

A typical pinball game will display the current score achieved so far and the number of balls that are left and when the game is over. The fewer balls there are left, the harder the player will try not to lose another ball, making the gaming experience more immersive.

Consistent Game Mechanics

Games should respond to users' actions in a predictable manner. Basic mechanics, such as hit detection and game physics, should all be appropriate for the situation. Game should react in a consistent, challenging and exciting way to the player's actions.
"HEURISTIC EVALUATION FOR GAMES: USABILITY PRINCIPLES FOR VIDEO GAME DESIGN" (2008)

The gaming experience is largely defined by its game mechanics. The player will expect and predict physically correct ball behavior. The player will to some degree predict the movement of the ball, so if the ball physics do not work consistently and physically correct the player will be confused and the user-experience will be degraded.

Provide intuitive and customizable input mappings

Controls should be intuitive and mapped in a natural way and leverage spatial relationships (the up button is above the down button, etc.) and other natural pairings. They should be customizable and they should also adopt input conventions commonly used in similar games.
"HEURISTIC EVALUATION FOR GAMES: USABILITY PRINCIPLES FOR VIDEO GAME DESIGN" (2008)

The game supports mouse, keyboard and a WiiMote to be used as input device. Due to the simplicity of the game there are only 3 actions the user can perform: starting the game and hitting each of the flippers. Using a WiiMote as input is however more complex as it allows arbitrary control of the flipper rotation via WiiMote rotation but should be self-explanatory nevertheless.

Discussion of Evaluation Results

The evaluation of the functional prototype shows that game mechanics and physics work perfectly. No problems have been pointed out in this regard. There was however a request for a score system. This request is easily understandable since the goal of most real-world pinball games is getting some kind of high score.

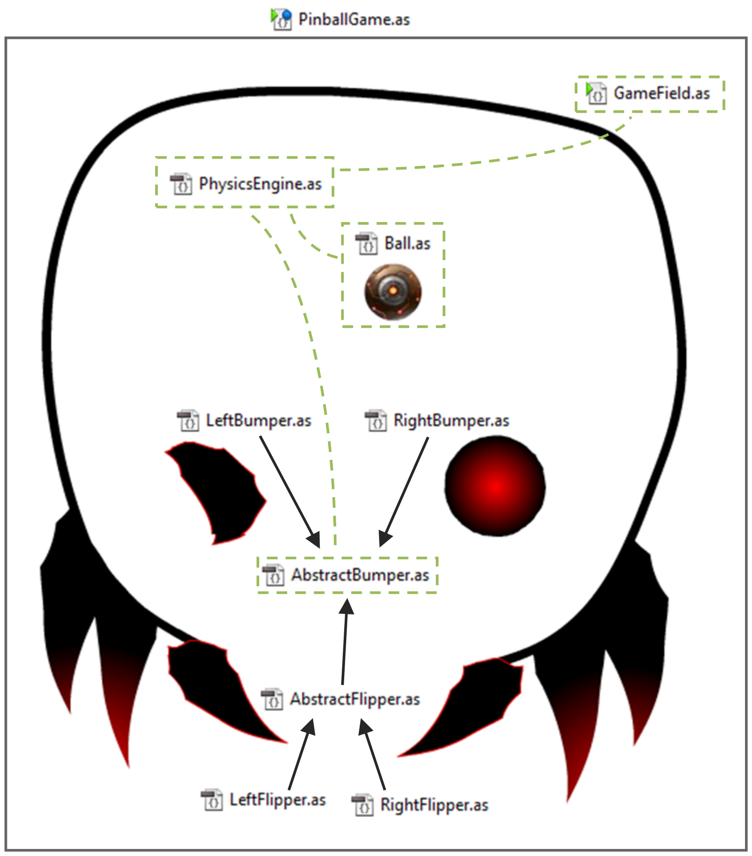
They only real issue that was reported is that there are no instructions as to how to play the game. This is however only a minor issue, as the controls are quite easily figured out. Additionally the controls are already active when the application is loaded, so the controls can be learned and tested before the ball is launched into the game.

Technical Evaluation

Architecture

Overview

The basic design principle of this application is that every visual object is represented by an ActionScript class. Inheritance is used to share common functionality and behavior (e.g. animation of the flippers).



Development Environment

The initial development environment was Flash CS4. This application appeals to designers because it is very easy to draw and animate on the timeline. Code editing however is rudimentary at best. Flash Builder 4 which was and used for development represents the exact opposite in terms of capabilities. It is an excellent IDE for developing ActionScript applications in terms of code, but has no support at all in terms of drawing and animation.

Both of these applications where used in development because of their respective strengths. Flash CS4 was used to create the visual assets which are then imported and used in the code that was written using Flash Builder 4. Exporting all the symbols from Flash CS4 as flash library (.swc) can be a bit tricky but once the library is added to the class-path in Flash Builder all the symbols that were exported as ActionScript classes can easily be instantiated and added to the stage.

```
var sprite:Sprite = new assets.metroid.Ball();
addChild(sprite);
```

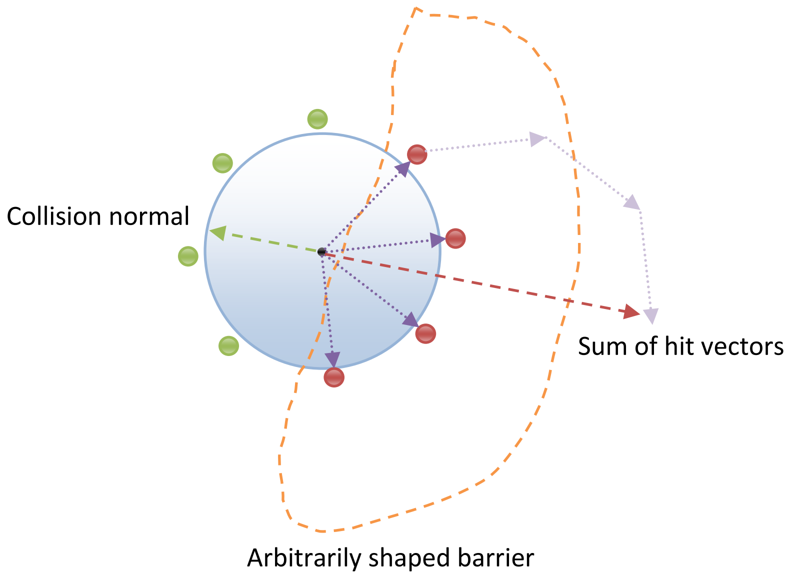
Ball Physics

The game consists of two major parts. There is scene graph for all the visual objects and the physics engine that animates the ball in respect to the other game elements.

The initial design concluded that a full-fledged physics engine is not within the scope of this assignment. A simple approach was chosen were every visual object is responsible for its own collision handling using internal properties such as rotation. That approach however would have been limited to rectangular and circular barriers. Arbitrarily shaped objects are not feasible for this and simplified more or less hardcoded way of collision handling.

During development however, it turned out that even though a full-fledged physics engine is out of the question, simplified ball physics can be implemented quite easily. The main problem in implementing physics is not collision detection itself, but in finding the collision normal, which is required to calculate the escape vector.

The perfectly circular shape of the ball allows for an easy solution. The sum all vectors originating from the center of the circle to a point on the perimeter of the circle that is in a state of collision, represents an average collision vector, which is the same as the collision normal except that it points toward the point of collision. The collision normal represents just a direction, so therefore the length of this vector is irrelevant.



```
private function getCollisionNormal(ball:Ball, barrier:Sprite):Vector3D {
    var cumulativeVector:Vector3D = new Vector3D();
    var radius:Number = (ball.width/2) - 4;

    for (var dg:int=0; dg<360; dg++) {
        var spot_x:int = ball.x + radius * Math.sin(dg * (Math.PI / 180));
        var spot_y:int = ball.y - radius * Math.cos(dg * (Math.PI / 180));

        if (barrier.hitTestPoint(spot_x, spot_y, true)) { // hitTest exact shape bounds
            var hitVector:Vector3D = new Vector3D(spot_x - ball.x, spot_y - ball.y);
            cumulativeVector = cumulativeVector.add(hitVector);
        }
    }

    if (cumulativeVector.length == 0)
        return null;

    cumulativeVector.normalize(); // normalize length
    cumulativeVector.negate(); // point away from the barrier
    return cumulativeVector;
}
```

The original design suggested that visual elements such as flippers and barriers are responsible for handling reflection of the ball themselves, meaning they calculate an escape vector for a given entry vector. Even though the reflection physics are now done by the physics engine each barrier element still gets a chance to post-process the escape vector that was calculated by the physics engine.

Different kinds of barrier elements like flippers and bumpers share a common abstract super-class. Implementations thereof may override the `updateEscapeVector()` to augment the default bouncing behavior.

A bumper element for example will increase the length of this escape vector, thereby increasing the speed of the ball. The flippers will not just increase the speed of the ball depending on their own state (moving up or moving down) but also adjust the direction of the ball. This adjustment, even though not physically correct, improves the gaming experience greatly, because the ball will always be reflected towards a desirable direction, even though the flipper didn’t hit the ball at a perfect angle.

```
override public function updateEscapeVector(momentum:Vector3D):Vector3D {
    if (currentSpeed > 0) {
        // flipper rotating upwards => add momentum when hit
        return momentum.add(new Vector3D(20 * currentSpeed, -70 * currentSpeed));
    }

    // if flipper is not moving
    return momentum;
}
```

Bounce and Roll Behavior

The section above describes how physically correct “bouncing off” can be simulated. In a pinball game however, the ball doesn’t always bounce off the flippers, bumpers or the outer wall of the game field. It sometimes just rolls down along the sides without bouncing off. “Rolling” can easily by achieved by continuously accelerating the ball while resolving collisions on every frame, thereby keeping the ball on its track.

```
// resolve collision
for (var v:Vector3D = normal; v != null; v=getCollisionNormal(ball, barrier)) {
    ball.x += v.x * 2;
    ball.y += v.y * 2;
}

// check if we should apply "bounce" or "roll" behaviour
if (ball.momentum.length > 5 || normal.y > -0.7) {
    // BOUNCE
    // get angle between entry and normal
    var entryAngleRad:Number = Vector3D.angleBetween(ball.momentum, normal);

    var matrix:Matrix3D = new Matrix3D();
    matrix.appendRotation(entryAngleRad * (180/Math.PI) * (-1), Vector3D.Z_AXIS); // reflection (rotate vector)
    matrix.appendScale(friction, friction, friction); // friction (scale vector)

    ball.momentum = matrix.transformVector(ball.momentum);
} else {
    // ROLL
    ball.momentum.x *= 1.2;
    ball.momentum.y *= 0.8;
}
```

Event System

The original design suggests a modular input system. Game mechanics and input listeners were supposed to be decoupled completely. This was not implemented however, as it would not affect that actual game play at all, but only increase development time.

Especially the physics engine would benefit from an event system and allow for better decoupling of the code. In the current implementation, the WiiMote control and the sound system are part of the physics engine and the physics engine itself is responsible for playing sounds and enabling the WiiMote rumble feature upon collision. This is clearly not part of the responsibilities of a physics engine class.

Future Prospects

One of the most obvious improvements is some kind of score system which was also a common point of criticism during evaluation. Adding a score system however is not be a particularly challenging task once collision detection is working, which is why it was omitted in the functional prototype.

Adding additional levels would also be quite easy. Since the collision detection is based on the shape of the game field and other visual elements, a new level could easily be drawn in Flash CS4 and then used in the application. Most of the work would probably be required to implement a graphically appealing selection menu for the different levels. The nature of the collision detection mechanism also allows for moving bumpers or even a moving game field which may make to game more interesting.