

**Balls!**

---

<b>COLLABORATORS</b>
----------------------

	<i>TITLE :</i> Balls!		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Jordi Fita	August 1, 2011	

<b>REVISION HISTORY</b>
-------------------------

NUMBER	DATE	DESCRIPTION	NAME
bf69c8e35b92	2011-08-01	The downloads for Balls are now images instead of text-only.	jfit
34b7522b4f97	2011-03-28	atangle is now using a new style for directives which don't collide with XML tags. I had to update all games and programs as well in order to use the new directive syntax.	jfit
6cc909c0b61d	2011-03-07	Added the comments section.	jfit
c9eaa8c0700e	2010-12-07	Adde Msc OS X download.	jfit
ee5e12ec551b	2010-12-02	Fixed a typo in an attribute in balls.	jfit
05834d60678b	2010-12-02	Added the download links for balls.	jfit
cbc64249ae63	2010-12-02	Added the ball's window title.	jfit
4cfce6ba4827	2010-12-02	Added the parameters to balls' main.	jfit
51b327de0ada	2010-12-01	Reworded some sections of balls.	jfit
f62268623456	2010-12-01	Reworded some of the sections in balls.	jfit
3dec1ee56026	2010-11-30	Fixed ball's introduction.	jfit

**REVISION HISTORY**

NUMBER	DATE	DESCRIPTION	NAME
ed9775e118c3	2010-11-29	Added the screenshot of balls.	jfita
d3c0c6c9668e	2010-11-29	Finished the first draft of balls.	jfita
83caede7f3ba	2010-11-26	Added the collision between explosions and balls.	jfita
7b0414a5b806	2010-11-25	Fixed bug with updating the Balls' direction.	jfita
b770974d9595	2010-11-25	Fixed a problem with clipping rectangles in Balls' SDL_FillCircle.	jfita
c7bbfb93e109	2010-11-25	Added Ball and Explosion classes.	jfita
df5df5644434	2010-11-24	Added the game loop section and drawing subsection.	jfita
11dd3e5908d0	2010-11-24	Added the SDL section to balls.	jfita
0fc39080c372	2010-11-24	Added the initial version of balls. It does nothing but create a bunch of levels.	jfita

---

## Contents

<b>1</b>	<b>Code</b>	<b>1</b>
1.1	Levels . . . . .	1
1.2	Balls . . . . .	4
1.3	Explosions . . . . .	6
1.4	Game Loop . . . . .	9
1.4.1	Frame Control . . . . .	9
1.4.2	Events . . . . .	10
1.4.3	Clearing the screen . . . . .	10
1.4.4	Removed Balls and Level Target . . . . .	10
1.4.5	Moving to the next Level . . . . .	11
1.4.6	Drawing the Start Screen . . . . .	11
1.4.7	Finishing . . . . .	12
1.5	SDL . . . . .	12
1.5.1	Initialization and cleanup . . . . .	12
1.5.2	The Screen . . . . .	13
1.5.3	The Window's Title . . . . .	14
1.5.4	Events . . . . .	14
1.5.5	Displaying Text . . . . .	14
1.6	Top Level Structure . . . . .	16
<b>A</b>	<b>Drawing Filled Circles</b>	<b>17</b>
<b>B</b>	<b>CMakeLists.txt</b>	<b>18</b>
<b>C</b>	<b>License</b>	<b>19</b>

---

## List of Figures

1	Screenshot of Balls! . . . . .	1
---	--------------------------------	---

---



Figure 1: Screenshot of Balls!

*Balls!* is a puzzle game in which the player must remove a level-specific number of bouncing balls from the screen. Balls are only removed when they touch an explosion's wave which, in turn, makes the ball explode and start a new wave. Clicking with the mouse on the screen, the player is only allowed to start a single explosion at the proper place and time which shall start a chain of explosions that must remove, at least, the required number of balls.

*Balls!* uses [SDL](#) and [SDL\\_ttf](#) and is written in C++.

[Download](#)[Download](#)[Source Code](#)

## 1 Code

### 1.1 Levels

Each level needs two values: the total number of balls on the screen and the number of balls to remove. This is stored in a structure.

```
<<level structure>>=  
struct Level  
{  
    unsigned int balls;  
    unsigned int target;  
};
```

Both the total number of balls and the target must be positive; it makes no sense to have less than zero balls.

*Balls!* has only 10 levels, for simplicity's sake, stored in an array which I initialize with the `balls` and `target` values.

```
<<levels definition>>=
Level levels[] =
{
    // balls, target.
    { 4, 1},
    { 4, 2},
    { 8, 4},
    {15, 7},
    {25, 12},
    {30, 17},
    {30, 20},
    {30, 25},
    {30, 28},
    {30, 30}
};
```

Then, instead of hard coding the number of levels in the array, I divide the array's size with the size of a single level to get the total number of levels. This has the advantage that to add a new level I only need to add it to the array without worrying about updating the number of levels available.

```
<<levels definition>>=
const unsigned int numLevels = sizeof(levels) / sizeof(Level);
```

Of course, the first level is 0.

```
<<levels definition>>=
unsigned int currentLevel = 0;
```

To initialize a level I create a new `vector` with as many balls as specified by the current level index in the array. Each ball has the same radius, but random position and color.

```
<<constants>>=
const unsigned int BALLS_RADIUS = 4;
```

```
<<initialize level>>=
std::vector<Ball> balls;
for (size_t ball = 0 ; ball < levels[currentLevel].balls ; ++ball) {
    balls.push_back(
        Ball(
            rand() % SCREEN_WIDTH,
            rand() % SCREEN_HEIGHT,
            BALLS_RADIUS, colors[rand() % colors.size()]
        )
    );
}
```

To be able to use `vector` I have to include its header file.

```
<<headers>>=
#include <vector>
```

The colors that can be assigned to the balls are converted from their red, green, and blue components using the screen's surface format and are kept in a `vector`. With this, I can select a random color by choosing a random index for that `vector`.

```
<<map level colors>>=
std::vector<Uint32> colors;
colors.push_back(SDL_MapRGB(screen->format, 255, 0, 0)); // red
colors.push_back(SDL_MapRGB(screen->format, 0, 255, 0)); // lime
```

```
colors.push_back(SDL_MapRGB(screen->format, 0, 0, 255)); // blue
colors.push_back(SDL_MapRGB(screen->format, 0, 255, 255)); // aqua
colors.push_back(SDL_MapRGB(screen->format, 255, 255, 0)); // yellow
colors.push_back(SDL_MapRGB(screen->format, 255, 0, 255)); // fuchsia
```

With this approach I avoid having balls with colors that are too similar to the background color or are too similar to each other. Fortunately, in *Balls!* the color doesn't have any impact on the game play.

Before I can use the random function, though, I need to set a *seed* value for the random generator. In this case, using the time in which the game started is good enough.

```
<<initialize random number generator>>=
srand(std::time(NULL));
```

I need to include the `cstdlib` header for `srand` and `ctime` for `time`.

```
<<headers>>=
#include <cstdlib>
#include <ctime>
```

In contrast to the balls, there are no explosions when the level starts. It is the player's duty to add a new explosion when she sees fit and start that chain of explosions from the removed balls. Thus, I create a new `vector` of explosions, but I don't add any until later.

```
<<initialize level>>=
std::vector<Explosion> explosions;
```

Besides balls and explosions, there are still two flags that the level must keep track of: one to know whether the level starter and another to know if the player already placed the first explosion on the screen.

The first flag is used to show a screen and wait for the player to click on the screen before the level starts. This is specially useful between levels, because otherwise the level would start right away after the previous level is cleared or when the player fails to meet the required target of balls to remove, in which case the message shown is slightly altered to tell the player to try again. This variable, thus, must be initialized to `false`.

```
<<initialize level>>=
bool levelStarted = false;
```

And is only set to `true` when the player clicks on the screen with the mouse.

```
<<process event types>>=
case SDL_MOUSEBUTTONDOWN:
    if (!levelStarted) {
        levelStarted = true;
    }
```

The other flag, to know whether the user placed the first explosion, is used to limit the number of explosions that the player can blow off. It is first set to the `false`.

```
<<initialize level>>=
bool firstExplosion = false;
```

Then, when the player clicks with the mouse button and the level has already started, this variable is set to `true` and a new explosion is added to the `vector`. If the player tries to click again, the game won't add another explosion to the `vector` until this flag is set to `false` again, which only happens when the next level starts or the player is forced to repeat the same level.

The explosion added to the `vector` when the player click the mouse button is set to the cursor's position, starting from a radius of 1 pixel and has a random color, like the balls.

```
<<process event types>>=
    else if (!firstExplosion) {
        explosions.push_back(Explosion(Vector2D(event.button.x, event.button.y), 1, colors[ ←
            rand() % colors.size()]));
        firstExplosion = true;
    }
    break;
```

## 1.2 Balls

A ball has four attributes:

1. Radius
2. Color
3. Position
4. Direction

The first two attributes, radius and color, are scalar values with the radius in pixels and the color as returned by `SDL_MapRGB`. On the other hand, position and direction are both 2D vectors. The direction attribute is used to update the ball's position when appropriate.

Thus, I need a 2D vector structure to hold the two coordinates,  $x$  and  $y$ , and a constructor to set the initial values of these coordinates. It would be possible to use `std::pair` instead, but the disadvantage of `std::pair` is that its two members are named `first` and `second` instead of the more familiar  $x$  and  $y$  for 2D vectors. To avoid confusion, I prefer to use a separate structure, which in this case doesn't need to be a template because I know that I only need integer 2D vectors.

```
<<2d vector struct>>=
struct Vector2D
{
    int x;
    int y;

    Vector2D(int x, int y):
        x(x),
        y(y)
    {
    }
};
```

With this new structure is now possible to define a new `Ball` class with the required attributes.

```
<<Ball class>>=
class Ball
{
public:
    <<Ball constructor>>

    <<Ball draw>>

    <<Ball update>>

    Uint32 color() const
    {
        return color_;
    }
};
```

```

    Vector2D position() const
    {
        return position_;
    }

    unsigned int radius() const
    {
        return radius_;
    }

private:
    Uint32 color_;
    Vector2D direction_;
    Vector2D position_;
    unsigned int radius_;
};

```

All the ball's initial attributes are passed as parameters to the constructor except for the direction, which is always initialized to move the ball upwards and to the left, although it would be possible to move in a random direction, for instance; it doesn't alter the game play at all.

```

<<Ball constructor>>=
Ball(int x, int y, unsigned int radius, Uint32 color):
    color_(color),
    direction_(-1, -1),
    position_(x, y),
    radius_(radius)
{
}

```

Once set, the radius and color never change within the lifespan of a `Ball` object, but each time the ball is updated, its position and possibly its direction changes. The position always changes according to the direction, but the direction only changes when the ball bounces on the level's limits. The level's limits are passed as parameter to the `update` function because the ball doesn't have to know anything else about the level and thus avoid coupling these two objects too much.

```

<<Ball update>>=
void
update(int minX, int minY, int maxX, int maxY)
{
    position_.x += direction_.x;
    if (position_.x <= minX) {
        direction_.x = 1;
    } else if (position_.x >= maxX) {
        direction_.x = -1;
    }

    position_.y += direction_.y;
    if (position_.y <= minY) {
        direction_.y = 1;
    } else if (position_.y >= maxY) {
        direction_.y = -1;
    }
}

```

To update all the all the position of the balls in the level, thus, I to iterate the vector of balls and call their update function passing the screen's resolution as upper limits and 0 as the lower limits, because the level uses all the screen.

```

<<update balls>>=
for(std::vector<Ball>::iterator ball = balls.begin() ;
    ball != balls.end () ; ++ball) {
    ball->update(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT);
}

```

To draw a ball on the screen I have to draw a filled circle of the specified radius, with the given color, and at the current ball's position. The surface on which to draw the ball is also passed as a parameter, because we could be drawing on a screen, on a back buffer surface, etc.

```
<<Ball draw>>=
void
draw(SDL_Surface *destination)
{
    SDL_FillCircle(destination, position_.x, position_.y, radius_, color_);
}
```

Much like updating, to draw all the balls on the screen, I iterate all the balls in the level's `vector` and call the draw function. In the case of `draw`, though, given that I only to pass a single parameter to the function, I can use the standard `for_each` to iterate the all the elements in the `vector` and the `bind2nd` structure to *bind* the screen surface to the draw second parameter, the first being the object itself. I also need to use `mem_fun_ref` to call a **member function** of an object **reference**.

```
<<draw balls>>=
std::for_each(balls.begin(), balls.end(), bind2nd(std::mem_fun_ref(&Ball::draw), screen));
```

The `for_each`, `bind2nd`, and `mem_fun_ref` helpers are declared in the standard algorithm header that I need to include.

```
<<headers>>=
#include <algorithm>
```

### 1.3 Explosions

An explosion also needs four attributes:

1. Color
2. Position
3. Radius
4. Radius growth

Three of the attributes are the same as the attributes for balls — color, position, and radius — but instead of moving around the screen, the explosions grows their radius up to a maximum and then shrink until the radius is zero. Whether the radius grows or shrinks is kept in the radius growth attribute, which is the number of pixels to grow or shrink the radius each update. Thus this value is a signed scalar.

```
<<Explosion class>>=
class Explosion
{
public:
    <<Explosion constructor>>

    <<Explosion draw>>

    <<Explosion update>>

    Vector2D position() const
    {
        return position_;
    }

    unsigned int radius() const
    {
        return radius_;
    }
}
```

```
private:
    Uint32 color_;
    Vector2D position_;
    unsigned int radius_;
    int radiusGrowth_;
};
```

The color and position are fixed through all the explosion's lifetime and passed as parameters to the constructor. The initial explosion radius is also passed as parameter, because it is different if the explosion is started by the player (the initial radius is zero) or if it is due a ball exploding (the initial radius is the same as the ball's radius.) In both cases, though, the explosion is growing and thus the growth rate is 1 pixel positive.

```
<<Explosion constructor>>=
Explosion(const Vector2D &position, unsigned int radius, Uint32 color):
    color_(color),
    position_(position),
    radius_(radius),
    radiusGrowth_(1)
{
```

As stated, when an explosion is updated it grows or shrinks depending on the sign of `radiusGrowth_`: if positive it grows, otherwise it shrinks. The explosion grows until it reaches a maximum radius, which passed as a parameter of `update`. Once it reaches this maximum radius, it then shrinks the radius until reaching 0, in which case the explosion is no longer active.

For that motive, I can't allow the `Explosion` class to have an initial radius of zero, otherwise it wouldn't grow at all. Passing a radius non-zero radius to the constructor is a caller's responsibility and I only enforce this condition using an `assert`. I could also use an exception, but I don't want to keep the condition for release build, where `assert` does nothing.

```
<<Explosion constructor>>=
    assert(radius > 0 && "The explosion radius can't be 0.");
}
```

The `assert` macro is defined in the standard `cassert` header file.

```
<<headers>>=
#include <cassert>
```

To update an explosion then I increment the radius as many units as specified by the `radiusGrowth_` if that radius is not zero already. The maximum radius that the explosion can grow to is passed as parameter.

```
<<Explosion update>>=
void
update(unsigned int maximumRadius)
{
    if (radius_ > 0) {
        radius_ += radiusGrowth_;
        if (radius_ >= maximumRadius) {
            radiusGrowth_ = -1;
        }
    }
}
```

For *Balls!* I've decided to use the same maximum radius for all the explosions. Thus, I need to define a maximum explosion radius as a constant.

```
<<constants>>=
const unsigned int EXPLOSION_MAXIMUM_RADIUS = 30;
```

Them, I use this constant to update all the explosions inside the level's `explosions` vector. If any of the explosions reaches a radius of zero, then this explosion gets removed from the vector.

```
<<update explosions>>=
for (std::vector<Explosion>::iterator explosion = explosions.begin() ;
     explosion != explosions.end() ; ) {
    explosion->update(EXPLOSION_MAXIMUM_RADIUS);
    if (explosion->radius() > 0) {
        <<check collision between explosion and balls>>
        ++explosion;
    } else {
        explosion = explosions.erase(explosion);
    }
}
```

If after updating an explosion its radius still isn't zero, then I need to check if it collides with any of the balls bouncing on the screen. If any ball enters in contact with then explosion, I replace the ball with an explosion by adding a new explosion to the explosions vector and removing the ball balls. Unfortunately, I can't just `push_back` the new explosion into the vector because if the vector needs to allocate more memory to accommodate the new element, then the iterator gets invalidated. To avoid these allocations, when starting the level I am going to reserve as many explosions as balls there are in the level, plus the explosion triggered by the user, because as the player can only start a single explosion there can't be any other explosion. Keep in mind that this doesn't *create* the Explosion objects, just reserve enough memory to hold at least as many Explosion objects as told.

```
<<initialize level>>=
explosions.reserve(levels[currentLevel].balls);
```

Without the possibility of allocations, now I can push the new explosion if a ball and explosion collides, remembering to remove the ball from the vector. The explosion's position, color, and initial radius are the same as the ball's.

```
<<check collision between explosion and balls>>=
for (std::vector<Ball>::iterator ball = balls.begin() ;
     ball != balls.end() ; )
{
    if (collides(*explosion, *ball)) {
        explosions.push_back(Explosion(ball->position(), ball->radius(), ball->color()));
        ball = balls.erase(ball);
    } else {
        ++ball;
    }
}
```

The collision function uses the fact that both the explosion and the ball are circles. Thus, if the distance between the centers of the two circles is less than the *sum* of their radius, the two are colliding. The distance between the two centers can be computed using the **Pythagorean theorem**, but instead of retrieving the actual distance, I am going to get the **squared distance**, to avoid a square root.

```
<<collision function between explosion and ball>>=
bool collides(const Explosion &explosion, const Ball &ball)
{
    unsigned int distanceSquared =
        (explosion.radius() + ball.radius()) * (explosion.radius() + ball.radius());
    return
        ((ball.position().x - explosion.position().x) * (ball.position().x - explosion. ←
         position().x) +
         (ball.position().y - explosion.position().y) * (ball.position().y - explosion. ←
         position().y)) < distanceSquared;
}
```

Finally, much like balls, to draw explosion I need to draw a filled circle at the explosion's position, using its current radius, and with the specified color.

```
<<Explosion draw>>=
```

```
void
draw(SDL_Surface *destination)
{
    SDL_FillCircle(destination, position_.x, position_.y, radius_, color_);
}
```

Again, to draw all the explosion in the level, I am going to use `for_each` but the explosions are drawn in reverse order, so newer explosions are drawn *under* older. This is because explosions are drawn after the balls and thus appear to be *on top* of them. When an explosion is fired, then, it also must come from *under* the already existent explosions. Drawing from the beginning to the end of the `vector` would, then, draw the explosions backwards of what is expected.

```
<<draw explosions>>=
std::for_each(explosions.rbegin(), explosions.rend(), bind2nd(std::mem_fun_ref(&Explosion:: ←
    draw), screen));
```

## 1.4 Game Loop

The game loop for *Balls!* is fairly typical: receive events, update entities, draw entities, update screen, limit the frame rate. This all must occur while the level is running which, in this game means until the player placed the first explosion and all the subsequent explosion have faded off; i.e., the vector of explosions is empty again.

```
<<game loop>>=
while (!quit && (!firstExplosion || !explosions.empty())) {
    <<process events>>
    <<clear screen>>
    if (levelStarted) {
        <<update balls>>
        <<update explosions>>
        <<draw balls>>
        <<draw explosions>>
        <<draw exploded balls and target>>
    } else {
        <<draw start level screen>>
    }
    <<refresh screen>>
    <<control frame time>>
}
<<check for next level>>
```

Notice how, as I said in Section 1.1, if the level hasn't started yet I do not update the balls nor the explosions and instead I show a screen that prompts the player to click a mouse button to start the level.

Also, when the level is finished, I need to check whether the player can go to the next level by checking if she has reached the target number of balls to remove.

### 1.4.1 Frame Control

As *Balls!* is a fairly low resource game, if I let the balls and explosions update as fast as the computer allows I would end up having a game too fast to be enjoyable. To avoid that, I pause for a few *milliseconds* each loop. Here I use the most easy and usually the less desirable way to control the game's frame per second: pause a fixed amount of milliseconds. In this case, I found that 50 milliseconds seems to be nice.

```
<<control frame time>>=
SDL_Delay(50);
```

This means that the game runs at a *maximum* of 20 frames per second (1000 / 50), but if the computer is not fast enough to cope with all the other chores it must perform within the loop, the game will run slower. This is the reason why this way is not too often used. In this case, being a very simple game, and without the need of very smooth animations I believe that it will be enough.

### 1.4.2 Events

For *Balls!* I only need to receive two kind of events:

- Quit
- Mouse click

The quit event is received when the player closes the game's window. In this case the only thing that I need to do is mark the game as over and end the loop.

```
<<process event types>>=
case SDL_QUIT:
    quit = true;
    break;
```

Another way to quit the game is by pressing the *Escape* key. This case is the same as the `SDL_QUIT`, but additionally I need to check if the pressed key is really the escape key.

```
<<process event types>>=
case SDL_KEYDOWN:
    if (SDLK_ESCAPE == event.key.keysym.sym) {
        quit = true;
    }
    break;
```

### 1.4.3 Clearing the screen

Once the level updated the status of all its entities — both balls and explosions — is time to draw them onto the screen. But, before drawing the *current* game's status, I need to remove the *previously* drawn state. The easiest way to clean up the previous state is to fill the whole screen with the background color before drawing any other entity. That is, of course, also the slowest method as it needs to fill regions of the screen that doesn't need to be redraw but most systems should cope with this, specially given the small game's resolution.

```
<<clear screen>>=
// The NULL rect is to fill the whole surface.
SDL_FillRect(screen, NULL, background);
```

The background color is an unsigned integer that I've extracted from the screen's surface by mapping arbitrarily selected red, green, and blue components.

```
<<map background color>>=
// 42 is always the answer.
SDL_Color backgroundColor = {42, 84, 126};
Uint32 background = SDL_MapRGB(screen->format, backgroundColor.r, backgroundColor.g, ↵
    backgroundColor.b);
```

### 1.4.4 Removed Balls and Level Target

While playing, the level must show the number of balls already removed and the number the level demands to be removed — its target — before passing to the next one.

The target is readily available from the `levels` array initialized at the beginning. To know the number of removed balls, instead of having a separate variable, I am going to subtract the number of remaining balls in the `balls` vector from the total number of balls that are supposed to be on the level.

```
<<draw exploded balls and target>>=
unsigned int explodedBalls = levels[currentLevel].balls - balls.size();
```

Then, using a string stream from the standard library, I convert this information to a string to draw onto the bottom right corner of the screen.

```
<<draw exploded balls and target>>=
std::ostringstream levelBalls;
levelBalls << explodedBalls << "/" << levels[currentLevel].target;
draw_right_string(font, levelBalls.str(), SCREEN_WIDTH - 10, SCREEN_HEIGHT - 20,
    fontColor, backgroundColor, screen);
```

The string stream is defined in the standard `sstream` header file.

```
<<headers>>=
#include <sstream>
```

### 1.4.5 Moving to the next Level

When the current level is finished, I need to check if the number of removed balls reached the target imposed by the level. Again, I can know the number of exploded balls by the difference between the remaining balls and the total number of balls. If this difference is equal or greater than the target, then I can increment the current level index.

```
<<check for next level>>=
nextLevel = levels[currentLevel].balls - balls.size() >= levels[currentLevel].target;
if (nextLevel) {
    ++currentLevel;
}
```

The `nextLevel` variable is set to `true` at the program's start. This variable is also used to know whether to show the "next level" screen or the "try again" screen when starting the level.

```
<<initialize variables>>=
bool nextLevel = true;
```

When the number of levels reached its limit, then the game must end.

```
<<check for next level>>=
finished = currentLevel >= numLevels;
if (finished) {
    quit = true;
}
```

The `finished` variable is initialized, as expected, to `false`.

```
<<initialize variables>>=
bool finished = false;
```

### 1.4.6 Drawing the Start Screen

The text to show while waiting for the user to click and start the level depends on whether we are really starting a new level or retrying the same. I can know this by looking at the `nextLevel` variable, which is set to `true` only when starting a new level.

```
<<draw start level screen>>=
std::string startLevelText;
if (nextLevel) {
    startLevelText = "Click to Start Next Level";
} else {
    startLevelText = "Oops! Try Again";
}

draw_centered_string(font, startLevelText, SCREEN_WIDTH / 2,
    SCREEN_HEIGHT / 2, fontColor, backgroundColor, screen);
```

### 1.4.7 Finishing

When all the levels are finished, the game shows a text informing the player.

```
<<draw finished text>>=
<<clear screen>>
draw_centered_string(font, "All Levels Finished. Well Done!", SCREEN_WIDTH / 2,
    SCREEN_HEIGHT / 2, fontColor, backgroundColor, screen);
<<refresh screen>>
```

Then, it waits for the player to close the window, press escape or click the mouse button. In this case I am using `SDL_WaitEvent` because I only need the event and don't need the CPU for anything else.

```
<<wait for exit event>>=
quit = false;
while (!quit) {
    SDL_Event event;
    SDL_WaitEvent(&event);
    quit = SDL_QUIT == event.type ||
        SDL_MOUSEBUTTONDOWN == event.type ||
        (SDL_KEYDOWN == event.type && SDLK_ESCAPE ==
            event.key.keysym.sym);
}
```

## 1.5 SDL

To use any function from SDL I first need to include its header. The actual directory in which the header is located at is passed to the compiler by the build system, thus including the `SDL.h` file (in caps, UNIX systems are case sensitive) is enough.

```
<<headers>>=
#include <SDL.h>
```

### 1.5.1 Initialization and cleanup

For this game, I only need SDL's timer and video subsystems. I tend to only initialize the SDL's subsystems that I really need instead of using `SDL_INIT EVERYTHING` to initialize everything, because if any of the subsystems can't be initialized — such as the CD-ROM under Dingux — the initialization would return an error. Specifying the subsystems that I *require* means that I don't need to which subsystem failed; I can't continue at all.

```
<<initialize SDL>>=
if (SDL_Init(SDL_INIT_TIMER | SDL_INIT_VIDEO) < 0) {
    throw sdl_error();
}
```

`sdl_error` is just a convenient class derived from `std::runtime_error` that uses the output of `SDL_GetError` to set the exception message to be the error as reported by SDL.

```
<<sdl_error class definition>>=
class sdl_error: public std::runtime_error
{
public:
    sdl_error():
        std::runtime_error(SDL_GetError())
    {
    }
};
```

`std::runtime_error` is a standard exception class declared and defined in the `stdexcept` header file.

```
<<headers>>=  
#include <stdexcept>
```

Once SDL is initialized, I need to make sure that SDL will be properly released when exiting the application. Being a pure C library, the best way to ensure that SDL is released when the application ends, either correctly or due an error, is to use the `atexit` function. With `atexit` I can tell the application to call `SDL_Quit` whenever the application calls `exit` either explicitly or by returning from the `main` function.

```
<<initialize SDL>>=  
atexit(SDL_Quit);
```

Another way would be to wrap the calls to SDL's functions in a class whose constructor would call `SDL_Init` and its destructor `SDL_Quit`, but to me it just adds unnecessary code for too little gain, although I have to admit that that using a wrapper class feels more like C++.

### 1.5.2 The Screen

With SDL initialized, I need a surface in which to draw the current game's status onto. Regardless of whether the surface is actually the whole system's screen or just a window, I will call this surface `screen`.

The screen's size for *Balls!* is set to the same resolution as an iPhone set in landscape mode, which is 480x320 pixels (HVGA.) This resolution is arbitrary and could be changed without any effect on the game play.

```
<<constants>>=  
const size_t SCREEN_HEIGHT = 320;  
const size_t SCREEN_WIDTH = 480;
```

As for the bits per pixel, I really don't care much. *Balls!* doesn't require a large number of colors, thus a 8-bit surface would do, but higher bits per pixel won't degrade the game's performance noticeably. Then, I'll tell SDL to use the current desktop's bits per pixel by specifying 0.

```
<<constants>>=  
const size_t SCREEN_BPP = 0;
```

The surface I request to SDL is a *software* surface, **not** hardware. According to SDL's documentation, to achieve a high framerate when doing per-pixel manipulation the screen, such as when drawing the circles, must be a software surface, otherwise each time the surface is locked the screen's contents must be copied from video memory to system memory and back when unlocked.

```
<<initialize screen>>=  
SDL_Surface *screen = SDL_SetVideoMode(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_BPP, ↵  
    SDL_SWSURFACE);  
if (NULL == screen) {  
    throw sdl_error();  
}
```

With this surface created, it is now possible to draw the game's contents on it. But, for the user to be able to see the actual screen I must update the screen's surface. The reason behind this update is that most systems, for performance reasons, don't update the application's contents all the time, only when necessary. What is deemed necessary is application dependent, but most application only needs to redraw the screen when a control, such as an edit box, changes its contents or when the application's window was previously hidden under another's window. This is, of course, not true for game which needs to update its screen each frame.

SDL provides the `SDL_UpdateRect` function to make sure that a surface's rectangle is updated and visible to the player, but I usually use `SDL_Flip` instead. When applying a surface without double buffering as a parameter, `SDL_Flip` actually behaves just like `SDL_UpdateRect(surface, 0, 0, 0, 0)`. But when the surface is double buffered it swaps the surface's back buffer and front buffer instead. Thus, `SDL_Flip` works for all surface and is easier to remember.

```
<<refresh screen>>=  
SDL_Flip(screen);
```

### 1.5.3 The Window's Title

SDL allows me to change the screen's window's title for my own instead of the default *SDL\_App*. I just need to pass the string to display on the window's top part and the text to show when the window is iconified. In this case I use the same string for both.

```
<<initialize SDL>>=  
SDL_WM_SetCaption("Balls!", "Balls!");
```

### 1.5.4 Events

In a typical SDL game, the application receives a number of *events* coming either from the player, such as a mouse click, or from internal subsystems, such as timers. SDL keeps all received events in a queue ready to be read by the game when it sees fit.

To get the events and remove them from SDL's queue, I am going to use `SDL_PollEvent`. SDL also provides the `SDL_WaitEvent` function to perform the same job, but the difference between the two functions is that `SDL_WaitEvent` waits **indefinitely** until there is an event, while `SDL_PollEvent` returns whether there is an event in the queue or not.

```
<<process events>>=  
SDL_Event event;  
while (SDL_PollEvent(&event)) {  
    switch (event.type) {  
        <<process event types>>  
    }  
}
```

### 1.5.5 Displaying Text

SDL by itself has no built-in functions to draw text on the screen. There are some different techniques to show text in SDL, but for simplicity's sake I am going to use [SDL\\_ttf](#). With `SDL_ttf`, I can open most TrueType fonts and use those to generate surfaces with the text on it.

All `SDL_ttf`'s functions are defined in the `SDL_ttf.h` header file.

```
<<headers>>=  
#include <SDL_ttf.h>
```

But before I can load fonts, I first must initialize the library in the same fashion as I initialize the base SDL library, except that I have no subsystems to initialize.

```
<<initialize SDL_ttf>>=  
if (TTF_Init() < 0) {  
    throw sdl_error();  
}
```

Just like SDL, at exit I need to call its clean up function.

```
<<initialize SDL_ttf>>=  
atexit(TTF_Quit);
```

Now is possible to load the font file by passing the font's file name and the size to use to draw text. For this game I am going to use [Bitstream Vera Fonts](#) due to their generous license.

```
<<load font>>=  
TTF_Font *font = TTF_OpenFont("VeraMono.ttf", 16);  
if (NULL == font) {  
    throw sdl_error();  
}
```

With the font loaded, drawing text is just a simple matter of calling the adequate SDL\_ttf's function to generate a new surface with the text to show. For *Balls!* I'll add two new functions to draw text. The first is to draw the font centered on a given position, the other to draw the text right aligned to the given position.

For the first function, I first need to generate the surface with the text by calling `TTF_RenderText_Shaded`. Then, I am going to use *half* the image's width and *half* the image's height to move the image's center to the specified position.

```
<<draw centered string function>>=
void
draw_centered_string(TTF_Font *font, const std::string &text, int x, int y,
    const SDL_Color &foregroundColor, const SDL_Color &backgroundColor,
    SDL_Surface *destination)
{
    assert(NULL != font && "The font is NULL.");
    assert(NULL != destination && "The destination surface is NULL.");

    SDL_Surface *textSurface = TTF_RenderText_Shaded(font, text.c_str(),
        foregroundColor, backgroundColor);
    if (NULL == textSurface) {
        throw sdl_error();
    }

    // Center the surface to the given position.
    SDL_Rect pos;
    pos.x = x - textSurface->w / 2;
    pos.y = y - textSurface->h / 2;
    pos.w = 0;
    pos.h = 0;

    SDL_BlitSurface(textSurface, NULL, destination, &pos);

    SDL_FreeSurface(textSurface);
}
```

The function to draw the right aligned text is almost the same, but instead of subtracting half the rendered text's width, it subtracts the *whole* width. The height is still centered, though.

```
<<draw right string function>>=
void
draw_right_string(TTF_Font *font, const std::string &text, int x, int y,
    const SDL_Color &foregroundColor, const SDL_Color &backgroundColor,
    SDL_Surface *destination)
{
    assert(NULL != font && "The font is NULL.");
    assert(NULL != destination && "The destination surface is NULL.");

    SDL_Surface *textSurface = TTF_RenderText_Shaded(font, text.c_str(),
        foregroundColor, backgroundColor);
    if (NULL == textSurface) {
        throw sdl_error();
    }

    // Center the surface to the given position.
    SDL_Rect pos;
    pos.x = x - textSurface->w;
    pos.y = y - textSurface->h / 2;
    pos.w = 0;
    pos.h = 0;

    SDL_BlitSurface(textSurface, NULL, destination, &pos);

    SDL_FreeSurface(textSurface);
}
```

```
}
```

In both cases I am going to draw the text in white.

```
<<define font color>>=  
SDL_Color fontColor = {255, 255, 255};
```

## 1.6 Top Level Structure

All the modules described above can be placed on a single source module ready to compile.

```
<<*>>=  
//  
// Balls! -- A simple game with balls.  
// Copyright 2010 Jordi Fita <jfita@geishastudios.com>  
//  
<<license>>  
//  
<<headers>>  
  
namespace  
{  
    <<constants>>  
  
    <<level structure>>  
  
    <<2d vector struct>>  
  
    <<sdl_error class definition>>  
  
    <<draw centered string function>>  
  
    <<draw right string function>>  
  
    <<draw filled circle function>>  
  
    <<Ball class>>  
  
    <<Explosion class>>  
  
    <<collision function between explosion and ball>>  
}  
  
int  
main(int argc, char *argv[])  
{  
    try {  
        <<initialize random number generator>>  
        <<levels definition>>  
        <<initialize variables>>  
        <<initialize SDL>>  
        <<initialize screen>>  
        <<initialize SDL_ttf>>  
        <<load font>>  
        <<define font color>>  
        <<map background color>>  
        <<map level colors>>  
  
        bool quit = false;
```

```

    while (!quit) {
        <<initialize level>>
        <<game loop>>
    }
    if (finished) {
        <<draw finished text>>
        <<wait for exit event>>
    }
    return EXIT_SUCCESS;
} catch (std::exception &e) {
    std::cerr << "Error: " << e.what() << "\n";
} catch (...) {
    std::cerr << "Unknown error\n";
}

return EXIT_FAILURE;
}

```

Notice how I wrapped the “happy path” in a `try catch` block that catches any thrown exception. All the exceptions thrown by *Balls!* are considered fatal, because I can’t do anything if I can’t initialize SDL, or the font, etc. Thus, I can only show the error to the user, using `cerr`, and then exit with an error status code.

To be able to use the `cerr`, I need to include the standard `iostream` header.

```

<<headers>>=
#include <iostream>

```

Also, even though I don’t use arguments in this application, I declared the `main` function to have the standard `int` and `argv *[]` parameters, instead of using an empty parameter list, which is also a possibility in C++. I’ve done this because SDL, in Windows and Mac OS X, renames the `main` function, for cross-platform uniformity, to `SDL_main` which *requires* those two parameters.

## A Drawing Filled Circles

I am using [Midpoint circle algorithm](#) but instead of only plotting point on the screen, I fill four rectangles per loop: two on the top hemisphere, and two on the bottom.

```

<<draw filled circle function>>=
void
SDL_FillCircle(SDL_Surface *dst, int cx, int cy, unsigned int radius, Uint32 color)
{
    if (0 == radius) {
        return;
    }

    int error = -radius;
    int x = radius;
    int y = 0;

    while (x >= y) {
        SDL_Rect rect;

        rect.h = 1;
        rect.w = 2 * x;
        rect.x = cx - x;
        rect.y = cy - y;
        SDL_FillRect(dst, &rect, color);

        rect.h = 1;
        rect.w = 2 * x;
    }
}

```

```

rect.x = cx - x;
rect.y = cy + y;
SDL_FillRect(dst, &rect, color);

rect.h = 1;
rect.w = 2 * y;
rect.x = cx - y;
rect.y = cy - x;
SDL_FillRect(dst, &rect, color);

rect.h = 1;
rect.w = 2 * y;
rect.x = cx - y;
rect.y = cy + x;
SDL_FillRect(dst, &rect, color);

error += y;
++y;
error += y;
if (error >= 0) {
    --x;
    error -= 2 * x;
}
}
}

```

## B CMakeLists.txt

*Balls!* is a cross platform game that needs to build and link against the [SDL library](#) on different platforms. A Makefile that would work on all configurations and platforms would be too cumbersome to write, specially when involving cross compilers.

There are numerous solutions available that make this process easier by detecting which platform to build to and configure the underlying building system accordingly. For *Balls!* I've chosen to use [CMake](#), a build system that relies on the system's appropriate tools (Makefiles, Solution files, etc.) to build and link the application, all this by using compiler independent configurations files.

CMake's configuration file is called *CMakeLists.txt* and the first thing I need to write is the project's name. The project's name is used, for example, as the name for Visual Studios Solution file.

```

<<CMakeLists.txt>>=
project(balls)

```

Although not strictly required, starting from version 2.6 CMake issues a warning if the configuration file doesn't specify the minimum required version of CMake. In this case, since I have only access to CMake's versions 2.6 and 2.8, I'll make version 2.6 the minimum required version. It is, of course, possible that this file works in earlier versions of CMake, but I have no way to test that.

```

<<CMakeLists.txt>>=
cmake_minimum_required(VERSION 2.6)

```

Being an SDL game, *Balls!* needs to tell the compiler where to look for its headers files as well as specify to the linker which libraries to link to. CMake includes a *package* that automatically looks for SDL's header and libraries files according to the build platform. Being a requirement, I tell CMake to stop if the SDL package can't be found by adding the `REQUIRED` parameter to `find_package`. The `SDLMAIN_LIBRARY` is required for Windows and Mac OS X only, but is harmless to add it for Linux.

```

<<CMakeLists.txt>>=
# SDL is required to build Balls!
find_package(SDL REQUIRED)
include_directories(${SDL_INCLUDE_DIR})

```

```
link_libraries(${SDL_LIBRARY})
link_libraries(${SDLMAIN_LIBRARY})
```

The same must be done with `SDL_ttf`.

```
<<CMakeLists.txt>>=
# SDL_ttf is required as well.
find_package(SDL_ttf REQUIRED)
include_directories(${SDLTTF_INCLUDE_DIR})
link_libraries(${SDLTTF_LIBRARY})
```

With all the build dependences found, it is now possible to build the game. Before that, though, I must extract the source code from the AsciiDoc document but only if the file is not already extracted. This is useful when distributing the game and don't impose a dependence to `atangle`.

CMake's `add_custom_command` is the ideal function to add a new command to the build system. In this case, I tell CMake that I want to generate a `balls.cpp` file from this document using `atangle` if `balls.cpp` doesn't exist or is older than the AsciiDoc document. Notice how I use the `{CMAKE_SOURCE_DIR}` variable to tell to CMake that the source file must be created in the same directory where `CMakeLists.txt` is and not in the directory where is building the game, which could be different.

```
<<CMakeLists.txt>>=
add_custom_command(
  OUTPUT ${CMAKE_SOURCE_DIR}/balls.cpp
  COMMAND atangle code.txt > balls.cpp
  MAIN_DEPENDENCY ${CMAKE_SOURCE_DIR}/code.txt
  WORKING_DIRECTORY ${CMAKE_SOURCE_DIR}
  COMMENT "Extracting aWEB source code")
```

Once extracted, this source code file can be used to build the executable. It is an habit of mine to add all source and header files in lists so I can later add properties onto. The list's name in this case is simply `SOURCES`.

```
<<CMakeLists.txt>>=
set(SOURCES ${CMAKE_SOURCE_DIR}/balls.cpp)
```

When CMake is analyzing the configuration file and adding build targets, it verifies that the files specified to build a target are available. In the case of *Balls!* the source code file could be missing if for example starting from a fresh checkout from the version control system and thus CMake would complain and exit before creating the necessary rules to extract the source code from the AsciiDoc.

Setting the file's `GENERATED` property to `ON`, CMake no longer verifies the existence of those files because as they are the output of some other tool — `atangle`, in this case.

```
<<CMakeLists.txt>>=
set_source_files_properties(${SOURCES} PROPERTIES GENERATED ON)
```

The last remaining bit of information to give to CMake is to tell it that I want to build a binary executable that needs to be a graphical windows application or Mac OS X's bundle, depending on the build platform, which the source files already listed.

```
<<CMakeLists.txt>>=
add_executable(balls WIN32 MACOSX_BUNDLE ${SOURCES})
```

## C License

This program is distributed under the terms of the GNU General Public License (GPL) version 2.0 as follows:

```
<<license>>=
// This program is free software; you can redistribute it and/or modify
// it under the terms of the GNU General Public License version 2.0 as
```

```
// published by the Free Software Foundation.  
//  
// This program is distributed in the hope that it will be useful,  
// but WITHOUT ANY WARRANTY; without even the implied warranty of  
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
// GNU General Public License for more details.  
//  
// You should have received a copy of the GNU General Public License  
// along with this program; if not, write to the Free Software  
// Foundation, Inc., 50 Temple Place, Suite 330, Boston, MA 02111-1307 USA
```