CMP105 Games Programming

Entity Management / Spawnables

This week

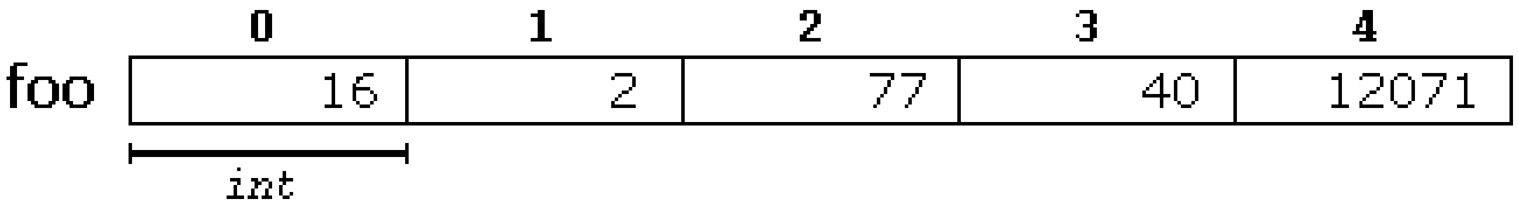
- Arrays
- Handling number of objects/entities
  - Different methods
  - Pros and cons
- Example
  - Building a management class
  - Spawning objects

Arrays, lists and vectors

- Array
  - Fixed-size
  - Supports fast random access
  - Cannot add/remove elements
- List (linked list)
  - Dynamic size
  - Supports bidirectional sequential access (poor random access)
  - Fast insert/delete at any point in the list
- Vector
  - Flexible-size array
  - Supports fast random access
  - Inserting or deleting elements other than from the back may be slow

Array

```
int foo[5];
foo[2] = 77;
```




```
for(int i=0; i<5; i++)
{
        // something
}
```

List

```
std::list<int> bar;
bar.push_back(10);
bar.push_back(100);

for (std::list<int>::iterator it = bar.begin(); it != bar.end(); it++)
{
        // something
}
```

Vector

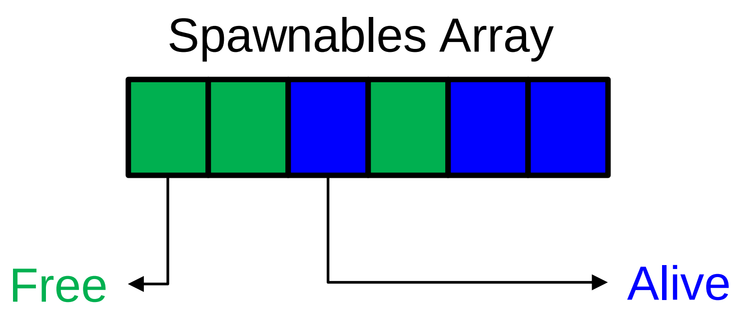
```
std::vector<int> foobar;
foobar.push_back(12);
foobar.push_back(120);

for(int i=0; i<foobar.size(); i++)
{
        // something
}
```

Entities / spawnables

- A number of games require game entities to be spawned and removed during gameplay.
  - Bullets
  - Enemies
  - Power ups
  - Etc.
- There are a number of different ways to implement this with their own pros and cons.

Method 1

- You can create an array with a size for the maximum number of spawnables.
- Each spawnable has a variable indicating whether it is alive or not.



- Spawning
  – Go through the array until you find a spawnable that is not alive.
  – Set this spawnable to alive and initialise it’s parameters (position, size, health, etc.)

Method 1

- Removing an object
  - Set the alive variable to false.
  - That's it ☺
  - This element in the array is now ready to be reused.

Method 1

- Updating

- FOR all elements in the array
  - IF the entity is alive THEN
    - Update the entity

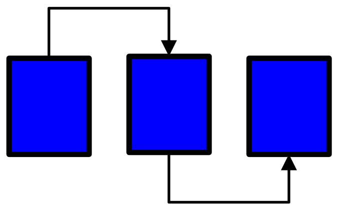
Method 1: Pros and Cons

- Pros
  - No memory allocation/deallocations occurs when spawnables are spawned/removed.
  - Simple to implement and maintain.
- Cons
  - Linear search to find a free spawnable when spawning
  - Excessive iteration over non-alive spawnables when updating or rendering
  - Searching time is linked to maximum array size
  - Possible over allocation of memory.

Method 2

- Linked list with Dynamic Memory Allocation
- A linked list of alive spawnables / entities

Array	Linked Lists
•Physically Contiguous •Fixed Length •Access Elements by Index •Insertion/Removal is Costly	•Logically Contiguous Only •Changeable Length •Access Elements by Traversal •Insertion/Removal is Efficient



Method 2

- Spawning

- Allocate memory for new spawnable / entity.
- Add it to the list

```
// declaration
std:list<Spawnable>::spawnables;
spawnables.push_back(Spawnable());
```

- push_back call allocates memory for new spawnable as well as maintaining the list.

Method 2

- Remove the spawnable from this list.
- Free up the memory.

```
for(
  auto spawnable = spawnables.begin();
  spawnable !=  spawnables.end();)
{
  if(remove spawnable test is true)
    spawnable = spawnables.erase(spawnable);
  else
    spawnable++;
}
```

Method 2

- Updating
- FOR all elements in the list
  - Update the respawnable

```
for(
  auto spawnable = spawnables.begin();
  spawnable !=  spawnables.end();
  spawnable++)
{
  // update the spawnable
}
```

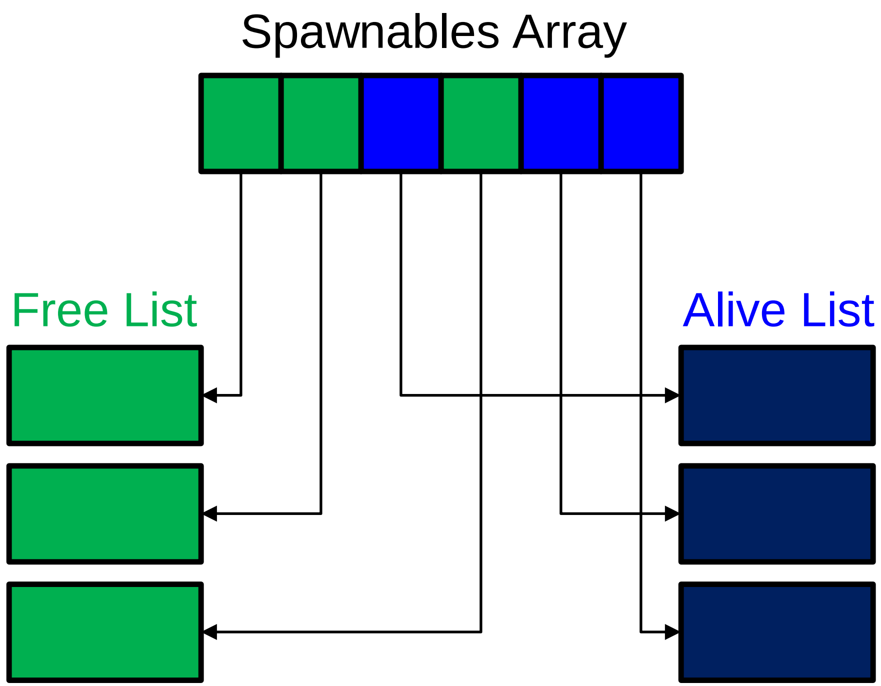
Method 2: Pros and Cons

- Pros
  - Only uses memory for alive spawnables.
  - Only iterate over alive spawnables when updating/rendering.
- Cons
  - Memory allocation/deallocation is slow when spawning and removing.
  - Maintaining list is a little more complex

Method 3

- Array with Lists
  - To avoid memory allocation, but only iterate over alive spawnables, use array to hold the spawnables and use lists to hold alive and free elements separately.

  - Create an array for maximum number of spawnables.
  - Create a list to store free (non-alive) spawnables and add all spawnables to this list.
  - Create a list to store alive spawnables. This will be empty on initialisation.



Method 3

- Spawning
  - Go to the free list and pop the first spawnable off.
  - Set the spawnable initial parameters.
  - Push it on to the alive list.
- Removing
  - Go to the alive list and find the spawnable to remove.
  - Remove the spawnable from the alive list.
  - Push it on to the free list.

Method 3

- Updating
  - FOR all elements in the alive list
    - Update the respawnable
- No point in updating dead ones

Method 3

- Pros
  - Only iterate over alive spawnables when updating/rendering.
  - No dynamic memory allocation.
    - If standard library containers are used there may be some memory allocation when maintaining lists.
    - This can be avoided if you implemented your own linked lists.
- Cons
  - Maintaining lists is more complex.
  - Possible over allocation of memory.

Considerations

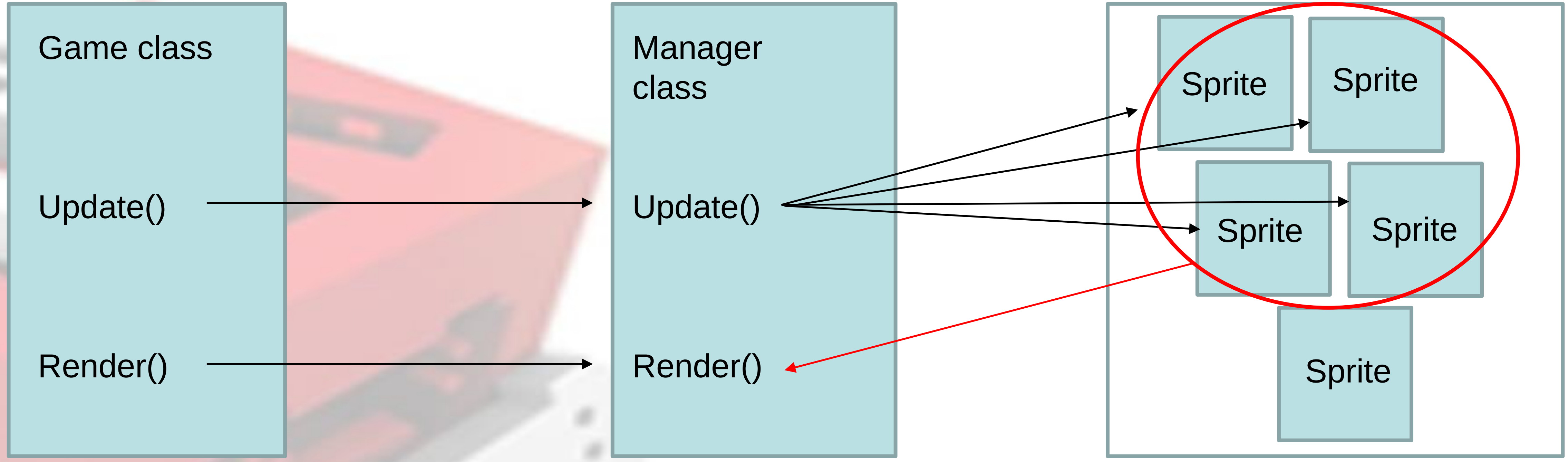
- There are still more methods that can be used to implement spawnables.
  - Custom memory allocators that are faster than standard library heap management.
- It's up to you to think what best suits your needs.
  - How often are potential allocations going to occur?
  - How tight is your memory budget?

Example

- Maintain a collection of sprites
  - Tracking if alive or not
  - Updating (position, etc)
    - Kill based on position (have they left the window)
  - Spawn sprite on key press

Manager class

- A class that manages the collection of objects
- Object stored in some form of list/array/vector
- Controls the sprites, so game doesn't have too



Sprite class

- Tiny update to sprite class
- Need a Boolean to track if sprite is alive (or not)

```
void setAlive(bool live) { alive = live; };
bool isAlive() { return alive; };

protected:
bool alive;
```

Ball class

- Very basic class
  - Inherits from sprite
  - Has basic movement code

Ball.h

```cpp
#pragma once
#include "Sprite.h"
class Ball : public Sprite
{
public:
Ball(const sf::Vector2f & size = sf::Vector2f(0, 0));
~Ball();

void update(float dt);

};
```

Ball.cpp

```cpp
#include "Ball.h"
Ball::Ball(const sf::Vector2f & size) : Sprite(size)
{
}

Ball::~Ball()
{
}

void Ball::update(float dt)
{
	move(velocity*dt);
}
```

Manager class

- Initialises a Vector of Ball class (all of the sprites)
  – Loads texture
- Update function
  – Loops over alive sprites and updates
- deathCheck function
  – Loops over alive sprites and check if they should die
- Spawn function
  – Finds dead sprite, re-positions it and sets it alive
- Render
  – Receives pointer to window
  – Loops over and renders, only alive sprites

```cpp
#pragma once
#include "Ball.h"
#include <math.h>
#include <vector>

class BeachBallManager
{
    public:
    BeachBallManager();
    ~BeachBallManager();

    void spawn();
    void update(float dt);
    void deathCheck();
    void render(sf::RenderWindow* window);

    private:
    std::vector<Ball> balls;
    sf::Vector2f spawnPoint;
    sf::Texture texture;
};
```

BeachBallManager.cpp

```
#include "BeachBallManager.h"

BeachBallManager::BeachBallManager()
{
	spawnPoint = sf::Vector2f(350, 250);
	texture.loadFromFile("gfx/Beach_Ball.png");

	for (int i = 0; i < 20; i++)
	{
		balls.push_back(Ball());
		balls[i].setAlive(false);
		balls[i].setTexture(&texture);
		balls[i].setSize(sf::Vector2f(100, 100));
	}
}

BeachBallManager::~BeachBallManager()
{
}
```

BeachBallManager.cpp

```
void BeachBallManager::update(float dt)
{
        // call update on all ALIVE balls
        for (int i = 0; i < balls.size(); i++)
        {
                if (balls[i].isAlive())
                {
                        balls[i].update(dt);
                }
        }
        deathCheck();
}
```

BeachBallManager.cpp

```cpp
// Spawn new ball
// Find a dead ball, make alive, move to spawn point, give random velocity
void BeachBallManager::spawn()
{
        for (int i = 0; i < balls.size(); i++)
        {
                if (!balls[i].isAlive())
                {
                        balls[i].setAlive(true);
                        balls[i].setVelocity(rand() % 200 - 100, rand() % 200 - 100);
                        balls[i].setPosition(spawnPoint);
                        return;
                }
        }
}
```

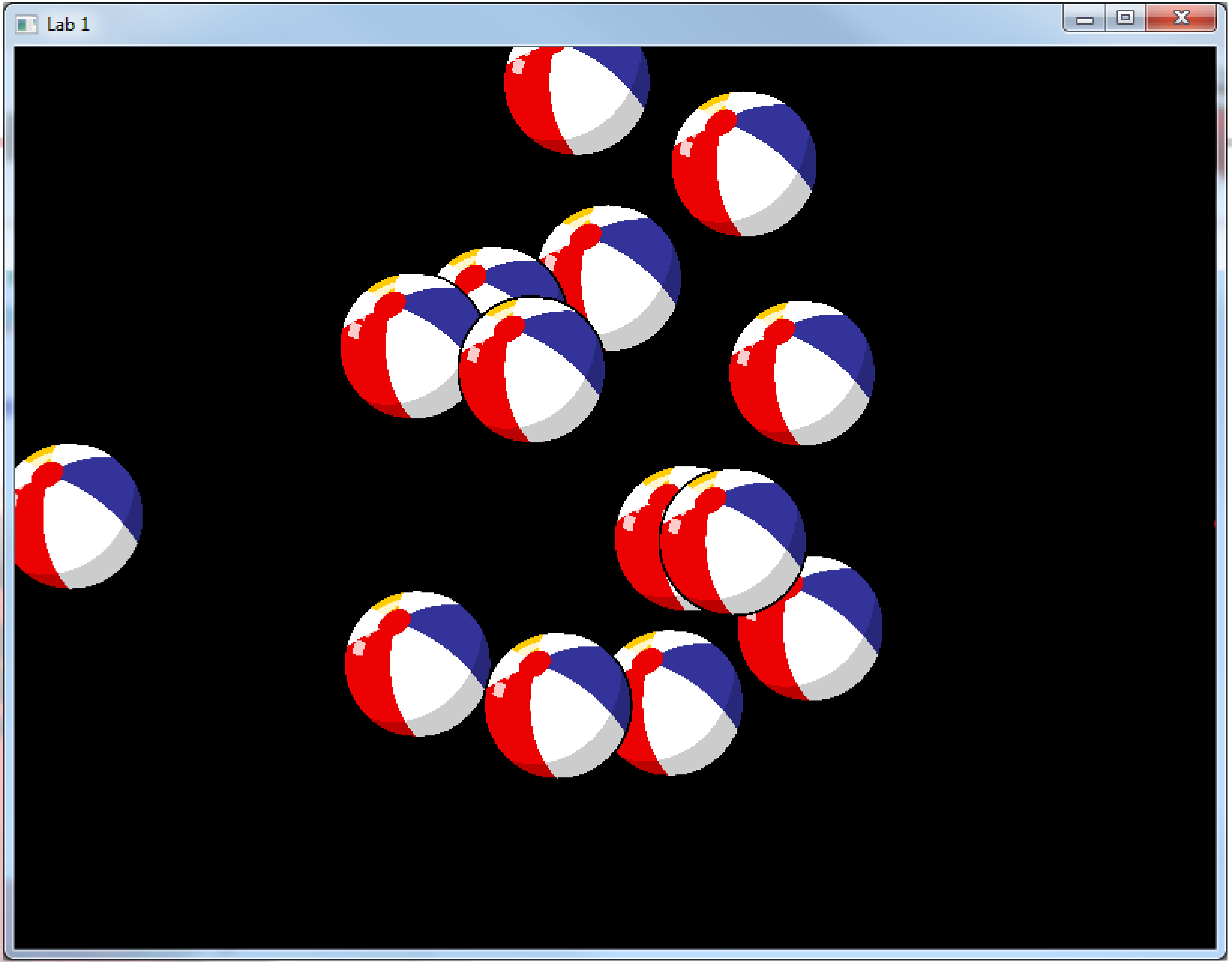
```cpp
// Check all ALIVE balls to see if outscreen screen/range, if so make dead
void BeachBallManager::deathCheck()
{
	for (int i = 0; i < balls.size(); i++)
	{
		if (balls[i].isAlive())
		{
			if (balls[i].getPosition().x < -100)
			{
				balls[i].setAlive(false);
			}
			if (balls[i].getPosition().x > 800)
			{
				balls[i].setAlive(false);
			}
			if (balls[i].getPosition().y < -100)
			{
				balls[i].setAlive(false);
			}
			if (balls[i].getPosition().y > 600)
			{
				balls[i].setAlive(false);
			}
		}
	}
}
```

BeachBallManager.cpp

```
// Render all alive balls
void BeachBallManager::render(sf::RenderWindow* window)
{
	for (int i = 0; i < balls.size(); i++)
	{
		if (balls[i].isAlive())
		{
			window->draw(balls[i]);
		}
	}
}
```

Live demo

- The sprite manager working


Lab 1

In the labs

- Building a manager class
  - Handling a large number of sprites

99 little bugs in the code,
99 bugs in the code,
1 bug fixed...compile again,
100 little bugs in the code.