What is ARPhysics

ARPhysics is a game-ready 2D rigid body physics simulation library. It was initially created as a final-year Computer Science project by Adrian Russell while at Plymouth University.
I should give massive thanks to Erin Catto & Scott Lembcke. As you may soon tell, a lot of the structure of this documentation is even based off of the documentation for Chipmunk Physics.

Limitations of ARPhysics

ARPhysics is designed for use in video games and so needs to be fast; taking over a second to work out how two bodies will interact is completely unacceptable. To achieve this requires the engine to only work with an approximation of physics. In other words the results of the engine are close to but not exactly correct to the real world and so is not useful for real world physical simulation.

How to compile it

- **Mac OS X:** The library comes with in an Xcode document to allow for easy compilation of the library and the demo application.
- **Linux:** To compile with linux you can compile using the supplied make file which will use gcc.
- W**indows:** At the current time to build for Windows you will need to create a Visual Studio Project that will compile all library code into a dll.

Contact

If you find any bugs, errors in the documentation, or have any questions or comment about ARPhysics then you can contact me at adrian.russell@students.plymouth.ac.uk

License

ARPhysics is licensed under the MIT licence.

This means that you can freely use & extend ARPhysics without having to purchase a license for commercial projects. (Though job offers or donations would be greatly appreciated.)

ARPhysics Basics

Overview

- **Worlds:** The space where the simulation of objects occurs.
- **SpatialIndexers:** The system that sorts out how objects are to be tested for collisions.
- **Bodies:** A body represents a single rigid body in the world and holds the physical properties (mass, position, rotation, velocity etc.). There are two types of bodies:
    - **CircleBody:** A body with a regular circle shape.
    - **PolygonBody:** A body with a convex polygon shape.
- **Force Generators:** Apply an impulse to a set of bodies each step. Useful for adding gravity or explosions to a simulation.
- **Constraints:** Constrain the movement of two bodies. Useful for creating springs and similar.

Memory Management

Most objects in ARPhysics that need to persist and be referenced inherit from an object simply named Object. This object adds basic reference counting and this is how memory is managed.

```
void retain()
void release()
```

When you instantiate a new object that is the subclass of object then you should always use the new keyword. This will create a pointer to that object and it will be given a reference count of 1. when you wish to destroy that object call the release() method and if it has not been retained again at any point its reference count will reach 0 and the object will delete itself. As this is C++ it is always possible to use the delete keyboard to immediately destroy an object however be warned that if you do so and another object has a reference to it then a crash can occur then that object attempts to access it. In other words, be careful and play by the rules of reference counting.

Basic Types

ARPhysics uses the standard primitive types supplied by C++ such as bool, float, int etc. to represent primitive types. Beyond those there are several other custom types to represent basic data:

- **Vector2:** a 2D vector type.
- **Rect:** a rectangle type.
- **Matrix22:** a 2x2 matrix type.

Basic Objects

Classes inherit from Object:

- **Array:** An array object that will retain and release objects when added to the array.

World

Worlds in ARPhysics are the space in which the physical simulation occurs.

To create a new world you first need to create a new Spatial Indexing system to manage the detection of body collisions. By default there are one type of partial indexing:

- **Brute Force:** The brute force indexing system does no optimisation for checking for collisions and all bodies are tested against all others. This system is acceptable for games with a small number of bodies.

To simulate the world you should call the `step(float dt)` method. This will simulate the world from its current state by the delta time you have specified in the function.

Spatial Indexing

The spatial indexing system is the part of the physics engine that controls the storing of bodies and the collision detection. This system is designed to be separate from the world to allow for the physics engine to be as flexible as possible for integration into games. You are able to produce any spatial indexing system you need to suit your needs from changing how the bodies are stored for collision detection (such as using a spatial hash or an AABB Tree) to what collision detection algorithms you use (such as a continuous collision detection system). Currently, the only spatial indexing system that ships with the library is a brute force indexing system that is suitable for games with fewer than about 100 dynamic bodies.

Creating a custom Spatial Indexing System

To create a custom Spatial Indexing system you need to subclass SpatialIndexing. Most methods that need to be implemented are self explanatory but two should be explained.

The bodies() method should return an Array of Bodies. There is no specified order that the bodies must be an and so may return a different order every time it is called, the World object only fetches this array once per step.

The `calculateCollisions(World*, float)` method must perform a series of action. This method should test for all collision and return an Array of Arbiter objects, one for each collision that should be resolved. The Spatial Indexing class you make must have some way to maintain the Arbiters of collisions that are in progress. For every potential collision that is to be tested, you must either get the existing arbiter for the collision or create a new Arbiter containing the two Bodies being tested. If a new arbiter has been created and a collision between the two bodies is detected then your indexing system should fetch the collision handler for the two bodies and then call the `begin(Arbiter*)` method. If this returns false then mark the Arbiter as being ignored then move on to the next collision test without returning that Arbiter. If true is returned then call the `preStep(Arbiter*)` method and then add it to the array. If there is an existing Arbiter and no collision then call the `seperate(Arbiter*)` method and do not add to the return array. Finally return the array.

This is somewhat complicated so look over how the `BruteForceIndexing` system works.

Below is the basic outline that your spatial indexing system should follow for each collision test in the `calculateCollisions` method.

```
// get the two bodies to test
Body *a, *b;

// if both bodies are static then skip
if (a->invMass() == 0.0 && b->invMass() == 0.0) {
     /** go to check next bodies **/
}

// if both bodies are in the same collision group then skip
if (a->collisionGroups() && b->collisionGroups() && (a->collisionGroups() & b-
>collisionGroups())) {
     /** go to check next bodies **/
}

// swap the types if necessary so that the body with smallest body type is first
if (a->bodyType() > b->bodyType()) {
     Body *c = a; a = b; b = c;
}

bool exists = true;

// get the arbiter if it exists
Arbiter *arbiter = /** Get existing arbiter or create new one if needed **/

// test to see if bodies collide, putting collision data into the arbiter
/** Some Collision detecting method with inputs (a, b, arbiter) **/

// get the collision handlers for the two bodies from the world
World::CollisionHandler handler = world->collisionHandlerForBodies(a, b);

if (arbiter->contact_count) {

     bool shouldBeIgnored = (arbiter->state == Arbiter::ArbiterStateIgnoring);

     if (arbiter->state == Arbiter::ArbiterStateNoCollision) {
             arbiter->state = Arbiter::ArbiterStateFirstContact;
             shouldBeIgnored = !handler.begin(arbiter);
             arbiter->state = Arbiter::ArbiterStateContact;
     } else if (!shouldBeIgnored) {
             arbiter->state = Arbiter::ArbiterStateContact;
     }

     // mark to ignore
     if (shouldBeIgnored) arbiter->state = Arbiter::ArbiterStateIgnoring;

     if (arbiter->state == Arbiter::ArbiterStateContact && handler.presolve(arbiter)) {
             /** Add arbiter to array to return **/
     }

     if (!exists) {
             /** add arbiter to store in spatial indexing implementation **/
     }
} else {

     if (arbiter->state == Arbiter::ArbiterStateContact || arbiter->state ==
Arbiter::ArbiterStateIgnoring) {
             // mark as separated and call separate collision callback
             arbiter->state = Arbiter::ArbiterStateSeparated;
             handler.end(arbiter);
     }
}

if (exists) {
     unsigned int seperationCount = arbiter->increaseStepCount();
     if (seperationCount >= /** number of steps to keep old arbiters **/) {
             /** remove from arbiter store in indexing implementation **/
     }
}
```

Bodies
Bodies are the rigid bodies that can interact in the system.

Static bodies are bodies that are fixed in place and are not affected by force from force generators or being impacted by other bodies. Static bodies are useful for creating floor, walls, and other fixed parts of terrain for makable bodies to interact with.
To set a body as static you can either set the mass to 0.0 or call the setStatic() method on the body.

Circle Bodies
A `CircleBody` is unsurprisingly a circular shaped body. The only additional information it stores beyond `Body` is the radius of the circle.

Polygon Bodies
Polygon bodies are convex rigid bodies. Beyond `Body` they contain a reference to a `PolygonPath` object. There are several methods to produce a new polygon body, one that takes a `PolygonPath` object as a parameter to allow multiple bodies to use the same path, which is totally allowed, and other which inter ally produce their own `PolygonPath`. If you are producing many objects that all the same shape it is recommended to produce a `PolygonPath` and share it among all bodies to keep your memory footprint low.

PolygonPath
A `PolygonPath` object which holds the geometric information about the shape of the polygon. When instantiated, the `PolygonPath` object performs a convex hull operation on the input vertices to ensure that they are convex; it then calculate the moment of inertia for the resultant shape and its area.

Arbiters
The `Arbiter` class represents the state of collision between two bodies. It contains data on how the bodies are colliding and can apply impulses to the two bodies to separate them.

An arbiter can have five states:
- **ArbiterStateNoCollision:** There is no collision between the two bodies. This is the state of a new arbiter before a collision has been detected.
- **ArbiterStateFirstContact:** The two bodies have collided on the current simulation step.
- **ArbiterStateIgnoring:** The arbiter has been marked so that the current collision between the two bodies should be ignored and not resolved.
- **ArbiterStateContact:** The two bodies are still colliding and are not being ignored.
- **ArbiterStateSeparated:** The two bodies have separated from colliding.

It should be clear by the NoCollision and Separated states that an Arbiter can also show that two bodies are not colliding. This is because it is often useful to keep an arbiter around after two bodies have separated for a number of steps to improve the stability of resting contacts what constantly oscillate between first contact and separation.There is a method called increaseStepCount() that will increment then return the number steps that have passed since two bodies separated. A spatial indexing system can then keep an arbiter and not restart a collision for a specified number of steps to improve stability (This is shown in the `BruteForceIndexing` system).

Force Generators
Force generators are used for adding force to all or a set of bodies in a world.

Supplied force generators

- **LinearForceGenerator:** Applies a specified impulse to a set of bodies
- **GravityForceGenerator:** Works in a way similar to a linear force generator except that the force is multiplied by the mass of the body so that all bodies will accelerate at the same rate
- **GravityToPointForceGenerator:** Accelerates a body towards a specified point in the world at a specified speed.

Creating new force generators

You can create your own force generators by subclassing the `ForceGenerator` class.

You need to implement the two virtual methods.

`bool shouldApplyToBody(Body *b)`
This method should return whether or not the force generator will be applied to the specified body. An example of the use is creating a force generator what works for a certain area in the world, this method could check if the specified body is inside that area and returning if the force generator should be applied.

`void applyToBody(Body *b, float dt)`
This method is where the impulse will be applied to the body. Only bodies that have been approved by shouldApplyToBody will be sent to this method. The delta time of the step is also passed to this method.

Constraints

A constraint is something that describes how two bodies affect each other. Constraints are useful for implementing structures like springs or rods or gears where two bodies are attached together and their movement are affected relative to the other.

Supplied constraints

- **Spring:** Acts like a spring.
- **DampedSpring:** Acts like a damped spring.
- **Rod:** Keeps two bodies a set distance apart.

Creating new constraints

You can create you own constraints by subclassing the `Constraint` class.
You just need to implement the `void solve(float dt)` method so that it applies impulses to both the bodies of the constraint. That's it.

Collisions

Collision Handling

There are four stages of collision callback that can be used to get information about a collision and affect the resolution of the collision. They work by registering a

```
typedef std::function<bool(Arbiter *)> CollisionBeginFunc;
typedef std::function<bool(Arbiter *)> CollisionPreSolveFunc;
typedef std::function<void(Arbiter *)> CollisionPostSolveFunc;
typedef std::function<void(Arbiter *)> CollisionEndFunc;
```

The begin collision callback is called during the first step of a collision. You can choice to ignore whether or not the collision should be resolved during the collisions life by returning a boolean

value. If you return false then the collision will be ignored and no further callbacks for this collision will be triggered until the bodies have separated and collide again.

The pre solve function is called before the collision is resolved in the current step. Again you can return false to not resolve the collision, although this time it will only be fore the single step. you can alter the forces, friction or restitution of the current collision in this callback.

The post solve callback is called after the forces have been resolved for the current step.

The end callback is called when the bodies have separated and the collision has ended.

Which collision handlers are used depend of the CollisionLevel of the bodies. If both bodies have no collision level or no custom collision handler handler has been set for those levels then the default collision handler is used. By default these are that all collisions will be resolved. You can overwrite the default collision handler to perform other actions.

You can set collision handlers that will be called only when certain levels of bodies collide. an example could be creating a level that repents a person with a collision level of, say, 1 and a level that represents bullet with level 2. You can then create a method that could register some form of damage to a body when there is a collision between a body with collision level 1 and a body with collision level 2.

```
#define CAR_LEVEL  1
#define WALL_LEVEL 2

/* Some method that adds a custom collision handler */
void someSetupMethod()
{
	// add a collision handler what will call bulletHit when a bullet object and a player object collide. Uses the default collision handler methods for all other callbacks.
	bool added = addCollisionHandler(CAR_LEVEL, WALL_LEVEL, crash, NULL, NULL, end);
	if (!added) {
		/** Handle not adding collision handler **/
	}
}

/* Method called when car first hits wall */
bool crash(Arbiter *arbiter)
{
	/** Do some actions; play sounds, mark as damaged etc. **/

	// carry out the
	return true;
}

/* Method called when car bounces off wall */
void end(Arbiter *arbiter)
{
	/** Do some actions about car bouncing off wall **/
}
```

In this example a collision handler is setup for car bodies and wall bodies, each with the corresponding collision level. When a car body and a wall body collides the crash method is called and when the bodies separate the end method is called.

Safely removing Bodies from the World

You should never remove a body from the world during a time step. Doing so could cause a bad access exception. to safely remove a body during a step you should set a post step callback that will remove body for you after the step has finished with it. You do this with the `addPostStepCallback(World*, void*)` method in world. Below is a trivial example of using this mechanism

```
#define PLAYER_LEVEL 1
#define BULLET_LEVEL 2

/* Some method that sets up a player and adds a custom collision handler */
void someSetupMethod()
{
	// add a collision handler what will call bulletHit when a bullet object and a player object collide. Uses the default collision handler methods for all other callbacks.
	addCollisionHandler(PLAYER_LEVEL, BULLET_LEVEL, bulletHit, NULL, NULL, NULL);
}

bool bulletHit(Arbiter *arbiter)
{
	// work out if body A or body B is player body by checking the collision levels
	Body *playerBody = NULL;
	Body *bulletBody = NULL;
	if (arbiter->bodyA()->collisionLevel() == PLAYER_LEVEL) {
		playerBody = arbiter->bodyA();
		bulletBody = arbiter->bodyB();
	} else {
		playerBody = arbiter->bodyB();
		bulletBody = arbiter->bodyA();
	}

	// some game specific code for example
	Player *player = playerForBody(playerBody);
	Bullet *bullet = bulletForBody(bulletBody);
	player->markAsHitByBullet(bullet);

	// add a callback to occur at the end of the step, it will cake removeDeadPlayer and will pass the player object as the key
	world->addPostStepCallback(removeDeadPlayer, player);

	// we don't want to continue resolving the collisions anymore
	return false;
}

void removeDeadPlayer(World *world, void *key)
{
	Player *deadPlayer = (Player *)key;
	Bullet *bullet     = deadPlayer->bullet();

	// remove the body of dead player and body of bullet from physics world
	world->removeBody(deadPlayer->body);
	world->removeBody(bullet->body);
}
```

In this example we have created a collision handler to deal with a bullet body (which is given the collision level, BULLET_LEVEL) hitting a player body (which has the collision level, PLAYER_LEVEL). When two bodies of these types collide the bulletHit method will be called by world and passed the arbiter for the collision. In this method we have worked out which body corresponds to the player and which the bullet by checking the collision levels of the bodies. In this example when the player is hit by a bullet he dies and we want both his body and the bullet body to be removed from the physics simulation. We can not just remove them immediately as there could be other bodies that want to perform a collision test against either of them so we must and do a post step callback which calls the `removeDeadPlayer` method, we have set the key to use as the player object because in this example it has a reference to both its own body and the bullet body

that hit it (convenient, huh?). As we are going to remove both of these bodies there is no point resolving the collision so we have returned false to ignore the collision. At the end of the current step of the physics simulation after all other bodies have been checked and resolved the post step callbacks that have been registered for this step are called including the one we did above. The removeDeadPlayer is called and passed both the world that called it and the key (the Player). At this point we remove both the player and bullet bodies from the simulation.

Setting up a new physics simulation

To set up a new physics simulation we must first create an instance of a spatial indexing system and assign it to be used by a new world instance. From this point we can add bodies, force generators and constraints to be simulated in the world.

```
#define WORLD_WIDTH  320
#define WORLD_HEIGHT 240

/* Some method that creates a simple world with a gravity that contains a ball. */
void someSetupMethod()
{
	// create an new spatial indexing system, in this case a brute force one
	BruteForceIndexing *indexing = new BruteForceIndexing();

	// create a new world. It will use the created spatial indexing system.
	World *world = new World(WORLD_WIDTH, WORLD_HEIGHT, indexing);

	// release indexing as the world retained it.
	indexing->release();

	// create a new body to put int the world.
	Vector2 circlePos    = Vector2(50.0, 50.0);
	float   circleMass   = 5.0;
	float   circleRadius = 5.0;
	CircleBody *circle = new CircleBody();

	// add that body to the world. It can now be simulated.
	world->addBody(circle);

	// release a reference to the circle body as the world has trained it.
	circle->release();

	// create a new force generator to simulate gravity.
	Vector2 gravityForce = Vector2(0.0, -9.81);
	GravityForceGenerator *gravity = new GravityForceGenerator(gravityForce);

	// add the gravity force generator to the world. This can affect the circle.
	world->addForceGenerator(gravity);

	// just like circle, we release as world has retained gravity.
	gravity->release();
}
```