Getting Started

- Introduction
- Installation
- Prerequisite Skillz
- Dependencies
- Resources
- Conventions
- Licensing
- Hello, qb2World!

Core Stuff

- Class Hierarchy
- Base Classes
- World Hierarchy
- Shapes
- Bodies
- Groups
- Joints
- Events
- Effects

Advanced Concepts

- Prototyping, Testing, Debugging
- Optimization
- Physical Properties
- Property Inheritance
- Extending the Library
- Low-Level Access
- Soft Objects
- Collision Filtering

Random Discussion

- Graphics
- Linked Lists vs. Arrays
- Units
- Troubleshooting
- FAQ
- Issues and Limitations

Introduction

QuickB2 is a wrapper of the popular Box2D physics engine. The name QuickB2 is a shortening of "Quick Box2D", and implies that you can do everything that you can using Box2D, but much quicker (in developer time that is). Its main goal is to wrap the low-level interface exposed by Box2D into something more intuitive, using proven paradigms like an extendable object-oriented architecture, nested hierarchies of objects, and the Flash event model. It is a complete abstraction, meaning no communication with the Box2D API is required.

Installation

The easiest way to get up and running is to import QuickB2/swc/QuickB2.swc into the IDE of your choice. This pre-compiled library wraps all the QuickB2 classes and their dependencies into one neat little package. If you want to compile off of source code, you must first include the QuickB2/src directory into your project, then the dependencies individually. For each dependency you have the choice of including .swc's or the source code.

Prerequisite Skillz

Good ActionScript 3.0 skills are necessary, as is good OOP know-how if you want to get the most out of things. It is a library for programmers, by programmers. Tools are included to empower artists and designers, but ultimately you'll need to write some code. You should also be familiar with basic physics and math concepts like forces, velocities, vectors, mass, transforms, etc., but it's not like you need a degree or anything. Experience with Box2D will definitely ease your entry, but QuickB2 is a complete abstraction meant to simplify the task of getting physics simulations up and running in the first place, so Box2D experience isn't necessary.

Dependencies

QuickB2 requires two libraries in order to compile, the Box2D Alchemy port of the C++ library (https://github.com/jesses/wck/tree/master/Box2DAS/), and an ActionScript 3.0 library called As3Math for its various geometry classes and math utilities. Box2D defines its own geometry/math stuff internally, but I use As3Math in so many other projects that it didn't make sense to rewrite all that code. These two dependencies come packaged so you don't have to venture out to find them, but they're not guaranteed to be up-to-date.

An optional library that's also included is the GUI/component library called MinimalComps (http://minimalcomps.com), which is required if you want to use qb2DebugPanel, a class facilitating runtime debugging by changing draw settings and spitting out stats. MinimalComps in general is a very good library to use for quick, light GUI's for debug purposes or prototypes.

You have the option of including either the pre-compiled .swc's or the source of the dependencies into your project. The dependency source code is pulled from the original repositories using the SVN "externals" property.

FlashDevelop is required if you want to build the examples. It's free, open-source, and awesome.

As3Math is the only dependency with which you'll have to work on a regular basis. Accordingly, documentation for this library is bundled with QuickB2's documentation.

Resources

Well, where to start. There might be a problem of too much information in this case. Besides this manual there's the API documentation, box2d.org with various manuals, wikis, and forums. World Construction Kit (http://www.sideroller.com/wck/) is an open-source project with similar goals to QuickB2 (abstracting Box2d, making it more accessible), but is catered more to non-programmers working with Flash CSx. The author of WCK also maintains the AS3 Box2D port that QuickB2 relies on. Examples are usually the best way for me to learn, and so I've included some with the library. Short and sweet readme's are also distributed liberally throughout the folder hierarchy.

For information on physics or math concepts, wikipedia is usually the go-to place for me. If you can't find the answer there, usually the reference section at the bottom of a page gives you what you need.

Conventions

All classes in QuickB2 have a "qb2" prefix to prevent naming conflicts with your code, increase code readability, and generally provide a cohesive feel to the library. You probably shouldn't use the "qb2" prefix in any extended classes you make for your own game or whatever, but it's not like I can stop you.

"Private" class members often use the qb2_friend namespace and are often suffixed with "_" to indicate that they're not supposed to be exposed to the outside world. A namespace access modifier is used instead of "private" because often QuickB2 classes have to communicate with each other behind the scenes, and there's no friend class construct like in C++. An average user doesn't even need to care about all this, but someone working with the source code will have to be aware.

Licensing

QuickB2 is released under the MIT (X11) license. Go here for more details, but it basically means you can do anything you want with it, as long as you don't try to say it's yours. This includes free commercial use. If you make it big using QuickB2, I congratulate you, but remind you not to forget the little guys! :)

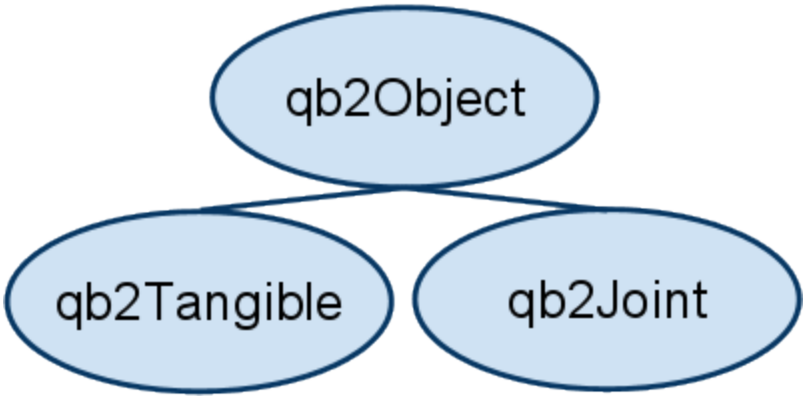
Hello, qb2World!

The following describes some common steps for starting a project using QuickB2...

1. Make a world, set the gravity, and set objects for debug purposes (drawing and dragging, and qb2DebugPanel) if you wish.
2. Set physical properties on the world that pretty much all objects will share in common, if any. For example, set friction to .5 if most objects should have .5 friction.
3. Add some groups to the world to organize things into logical chunks. For example, make a group for the walls in your level, for enemies, for objects the player should interact with, etc.
4. Set properties on the groups just like you did for the world.
5. Add shapes, bodies, and even sub-groups to the groups. These will constitute individual game elements like the player, enemies, walls, etc.
6. Attach actors to your shapes and bodies. Bitmaps generally offer the highest performance, at the cost of RAM and file size.
7. Fine tune any physical properties you want on the individual game elements.
8. Add event listeners to listen for things like contacts (e.g. to play a sound when a ball hits a wall), adds/removes, updates, etc. Most of your game logic will be written inside event handlers.

9. Call qb2World::start() and watch the fun.

Class Hierarchy



Base Classes

The following serve as base classes for all physics objects.

- qb2Object serves as the base class for all physics objects.
- qb2Tangible extends qb2Object and is the base class for all objects that have mass, geometry and other physical properties. They are the actual "things" that you interact with in the world.
- qb2Joint also extends qb2Object and is the base class for all joints, which constrain qb2Tangible objects to move in interesting ways.
- qb2ObjectContainer extends qb2Tangible and is the base class for qb2Body and qb2Group. You can add any qb2Object to a qb2ObjectContainer instance.
- qb2Shape also extends qb2Tangible and is the base class for qb2PolygonShape and qb2CircleShape, which are used to define the geometry for your qb2Tangible objects.
- qb2IRigidObject is technically an Interface, buts acts more as another abstract base class for qb2Body and qb2Shape. If ActionScript 3.0 allowed multiple inheritance, qb2IRigidObject would in fact be made into a class. It gives an object such things as position and rotation, and allows it to be attached to a joint.

You never instantiate any of the above classes directly. In fact doing so will throw an error. The frequency of their direct use is related to how far down the list they are, i.e. you rarely use qb2Object directly. Besides their OOP-ness, they exist to handle a lot of internal stuff that you generally don't need to worry about.

World Hierarchy

Every physics world you make forms a tree with a qb2World at the root, branches made of qb2ObjectContainer's, and qb2Shape's and qb2Joint's at the leaves. You cannot add a world to another world. Generally you will/should only make one world for your application. Some functions declared by qb2ObjectContainer act as ways to modify the world tree. qb2ObjectContainer::addObject()/ removeObject() are the most basic ones. You also have ways to traverse the tree. qb2Object::parent/ world allow you to go up, and qb2ObjectContainer::getObjectAt()/numObjects allow you to go down. The API is similar to the one used by flash.display.DisplayObjectContainer and many other engines featuring display hierarchies.

Your typical set-up will look something like a world containing groups containing bodies containing shapes, but you can make any kind of tree you want. Complex objects, like let's say a rope made of joint-connected circles, can be a group containing joints and shapes. The rope is then one modular object that can be added/removed to/from the world or some other group, instead of a bunch of individual objects.

Shapes

Shapes are your basic building blocks, and come in two varieties: qb2CircleShape and qb2PolygonShape. They implement qb2IRigidObject, and can be added directly to the world without wrapping them in a body, as Box2D requires. Joints can be attached to them, and you can add shapes to bodies in order to make more complex rigid objects. The qb2Stock class has a nice suite of functions for creating common shapes.

Bodies

qb2Body serves much the same function as b2Body in Box2D; that is, it allows you to wrap a bunch of shapes together in order to make more complicated objects. For example, making a rounded rectangle *could* be done with just one polygon shape, but this would be very inefficient and tedious to program. It's much easier to construct one out of four circles and three rectangles.

Groups

A group is like a qb2Body except its objects can move around freely. Thus it doesn't have a position or rotation. You can use qb2Groups to make "soft objects", or organize your game's entities into logical chunks.

Joints

Joints are used to constrain two rigid objects (shapes or bodies) relative to each other in some way. For example a revolute joint constrains two objects to rotate around a common axis for things like wheels, while a distance joint keeps two objects a certain distance apart. Joints can be added to the world at any time, and are kept in the world until they or a parent is explicitly removed (this behavior differs from Box2d, where a joint is implicitly removed if an attached object is removed). They only ever do something however when both the objects they are meant to constrain are also in the world. All joint types require two rigid objects with the exception of a mouse joint, which requires just one object and uses a point (usually mouse position) as the second constraint. All joints use joint anchors to define exactly where an object is constrained. These anchors are ultimately set relative to the local coordinate space of the attached object, but convenience functions using world points are also provided.

You can add joints anywhere you want in the world hierarchy to attach objects residing anywhere in the world hierarchy. For example you might choose to add all joints directly to the world, regardless of which sub-groups the joints' attached objects reside in. In practice however, it's best to add joints to the closest ancestor that contains the objects that the joint is attaching together, just to keep things well-organized.

Unlike Box2D, joints can reside in the world without their rigid objects being in the world also; the joint will become active automatically when its objects are added. This means you can add a joint to the world first if you like, then its objects later if your setup calls for it.

Events

QuickB2 uses the Flash event model because it's flexible, intuitive, and widely adopted. This is somewhat at the cost of performance, as there are certainly more efficient ways of reporting events. However, QuickB2 internally tracks which objects have which listeners attached using fast bitwise operations, so events are only dispatched when they need to be. Further, actual event instances are cached for every type of event, limiting memory overhead significantly compared to if a new event was instantiated every time it was needed.

You can listen for a variety of events in QuickB2, such as when an object is added or removed, when it's updated, when its mass or other physical properties change, and a few others. The most common ones you'll probably listen for though are contact events. QuickB2 has a significant advantage over Box2D in that the world can be changed in any way you like within a contact handler. You can add, remove, and transform any object instead of having to set up flags and handle things after the time step. Experienced Box2D users will note that this is impossible, and indeed it is. The workaround is that if QuickB2 receives a command inside a contact callback to change an object in a way that is not allowed in Box2D at that time, the command is put in a queue until it's safe to process it. This is all transparent to the user however, and if you have no idea what I'm talking about, don't worry about it.

Effects

Effects are used to add interesting procedural movement to the qb2Tangible objects in your simulation, and thus serve much the same purpose as Controllers in Box2D. For instance you can use qb2Wind to either simulate air resistance or blow objects around your screen. A qb2Vibrator effect will make an object shake around randomly. All effects can be adjusted parametrically to suit your needs, or you can extend the classes provided to either override or adjust things to taste. You can either add a qb2Effect to a qb2Tangible's 'effects' array, where it will be applied automatically every time step (the preferred method), or you can apply an effect manually yourself through qb2Effect::apply(). You can kind of think of an effect as the physics-engine equivalent of a flash.filters.BitmapFilter in Flash, if that makes sense.

Although any effect can be applied to any tangible, some effects are only really applicable to qb2Group instances. For instance qb2PlanetaryGravity is an effect that makes objects orbit around each other like planets, and for that you need more than one object. Applying this effect to a qb2Shape or a qb2Body wouldn't do anything.

Effects are reusable. That is, one effect can be applied to any number of objects, and reside in multiple objects' 'effects' arrays.

Prototyping, Testing, Debugging

There exist various utilities packaged with QuickB2 that let you get up and running quickly, and solve issues quickly...

- qb2Object::drawDebug() can render your objects for you until you have a good graphics pipeline in place, and can help resolve issues of misalignment between graphics and physics later on down the road.
- qb2DebugDrawSettings globally controls how and what qb2Object::drawDebug() will render.
- Setting qb2World::debugDrawContext gets debug drawing up and running in one easy step. All objects residing in the world will automatically call drawDebug() to render to this Graphics context.
- qb2DebugPanel provides a graphical interface that can be used to adjust certain properties of qb2DebugDrawSettings, and displays helpful stats like FPS and numbers of different objects in the world. This tool requires the MinimalComps library (http://minimalcomps.com).

- Setting qb2World::debugDragSource allows your world to automatically manage a mouse joint based on Flash mouse events, allowing you to interactively drag objects around. As the name implies, it's not meant for actual production-level mouse controls.
- All the core classes in QuickB2 spit out verbose, customizable information for trace() calls.
- qb2DebugTraceSettings defines the variables that are displayed for certain classes when you do a trace(), and how they are displayed. These settings can be changed to your taste either at runtime or compile time.
- qb2Keyboard wraps Flash keyboard events to provide an API more useful for games, with functions to check if a key is down, the last key down, and so on. It also only fires one KEY_DOWN event instead of a stream.
- qb2Mouse similarly wraps Flash mouse events to provide a more game-centric mouse API.

Pretty much all these (with the exception of qb2Keyboard/qb2Mouse) add unnecessary overhead to your code, and so probably shouldn't be used in final release versions.

Optimization

Here are some optimizations you can consider once you have a beta going and are ready to lose some fat. Most of these are pretty minor, but even the minor ones can add up.

- Use simple CCW convex polygons with as few sides as possible. These is extra overhead involved in both initializing and simulating non-standard polygons.
- Try to use polygons with 8 or less vertices. More than this results in overhead similar to having to deal with non-standard polygons.
- Setting qb2Settings.checkForNonStandardPolygons to false will result in a small performance gain any time a polygon is added to the world, but it means that all your polygons must be simple, CCW, convex, non-self-intersecting, and have less than 8 vertices.
- Wrap dynamic polygons in bodies for a small performance benefit, as the points of the polygon don't have to be repositioned every frame. Polygons that are stationary or rarely moved (like those making the floors of a level) don't have this extra cost.
- Use circles whenever possible. They're easier to manage for the engine in really every way.
- Changing the geometry or mass properties of a shape/body while it's in the world costs more than when it's outside the world. Therefore try to set this stuff up before adding things to the world.
- Add multiple objects to a container at once (either through qb2ObjectContainer::addObjects(), or addObject() with multiple arguments). This consolidates the work that has to be done internally every time an object is added to a container.
- Avoid excessive adding/removing of objects from the world. Instead of removing a body and readding it later, you could try setting it as a sensor, putting it to sleep, changing its contact filter so it doesn't collide with anything, etc.
- Optimize graphics. What may seem like a choke in the physics pipeline may actually be a rendering problem. Google stuff like "flash graphics optimization", but basically vector graphics are big performance hogs, particularly lines, whether it be through the Graphics API or prebuilt. Bitmaps almost always perform better, usually at the cost of higher RAM/filesize.
- Store local copies of particular qb2Object properties that you need to access repeatedly. General rule of thumb is that any property/function return value that you get that is a type of amEntity2d should be cached (so amVector2d, amPoint2d, amBoundBox2d, amBoundCircle2d, etc.).
- Make your contact handler functions as fast as possible. These handlers can get called several times per step.
- Avoid excessive nesting of containers. A few levels deep isn't a problem, but if you go over like 10 levels deep, you're probably doing something wrong.

- Debug drawing isn't particularly optimized in any way. Its purpose is for, well, debugging...relying on it for final graphics might not be a good idea.

Physical Properties

TODO

Property Inheritance

Tangible objects in QuickB2 inherit certain property values from their parents by default. For example, if you create a world, and give that world a friction of .78, all subsequent objects added to the world will have a friction of .78 as well, including objects added to objects added to the world, and so on. However, if you explicitly set the friction to .9 on a certain object, let's call it myBox, myBox (and all its children if it's a container) will continue to use .9 after it's added to the world, overriding the world's friction. However, if after adding myBox to the world, you again set the world's friction to .78, myBox's friction will become .78. Simple, right? Well don't worry, it sounds complex, but it's like a car, complicated on the inside, but easy to drive.

This system is useful because it's very rare that you want every object in your simulation to have wildly different properties. It's even rarer to have a desire to set these properties individually for every object in the world, i.e. it's quite annoying to have to do so. Property inheritance therefore keeps your code very clean, and helps with the whole housekeeping side of managing your physics simulation.

Extending the Library

Providing an extendable architecture is one of the main goals of QuickB2. Physics simulation is a prime candidate for good object-oriented programming, and making your own class or even library extensions is highly encouraged. The classes you will most often extend are qb2Body and qb2Group. These classes play a similar role to Sprites or MovieClips in Flash. Extending any other type of qb2Object would be extremely rare, and is discouraged unless a good reason is brought to my knowledge.

The qb2Effect classes are also completely, easily, and encouragingly extendable.

As best practice (although I hate that phrase), any extended classes you make should not require any parameters in their constructors. Parameters with default values are usually fine though.

Remember to call the super function of any functions/constructors you are overriding.

Low-Level Access

In some cases access to the Box2D API is nice to have; working around a bug, using some feature that isn't explicitly supported (yet) in QuickB2, or optimizing some operations, to name a few. This API can be accessed through qb2World::b2_world, qb2Joint::b2_joint, qb2Shape::b2_body, qb2Shape::b2_fixtures, and qb2Body::b2_body. However, **only use this API to read data and make queries**. Making any kind of changes to Box2D objects through these side doors can really wreak havoc. Besides b2_world, your QuickB2 object will have to be in a qb2World for the property to be non-null.

Note that the m_userData properties of Box2D objects will point to their QuickB2 counterparts.

Soft Objects

Soft objects, more commonly called soft bodies, are not supported explicitly in QuickB2, because QuickB2 wraps an exclusively rigid-body engine, Box2D. However, using groups, joints, and a clever rendering technique, you can get a very good approximation. qb2SoftPoly is the best example of this. "Soft objects" in QuickB2 don't have positions or rotations since these properties aren't well-defined. It would be like saying a jiggling piece of jello is at such and such a position in a room...it can't be done precisely because the jello moves relative to itself. Soft objects do however have most of the properties and methods of rigid objects like qb2Body and qb2Shape since they all inherit from qb2Tangible, allowing you to treat them as one distinct object for the most part, ignoring the nuts and bolts that make them up.

Using the 'centerOfMass' property is a good way to approximate the position of a soft object, but must be calculated from frame to frame, so this can get expensive for complicated objects.

Collision Filtering

TODO

Graphics

In principal QuickB2 is agnostic to rendering; you can handle graphics any way you want, and there's no "best practice". However, for the ActionScript 3.0 version, every tangible object has an actor property in the form of a DisplayObject, and if set, QuickB2 will pull some basic strings for you, namely setting x, y, and rotation every frame, and adding/removing the graphic from its parent whenever the object is added/removed. This is a very simple and non-invasive technique, but actually ends up being robust enough for most games, so much so that as long as you set up the actors initially in a good way, you usually need never worry about game graphics after that.

Every object inheriting from qb2Object also implements qb2IDrawable, declaring draw() and drawDebug() functions to facilitate drawing of runtime vector graphics. For the most part these functions act as a way to quickly see your simulation in action, and usually shouldn't be used for final rendering. The reason for this is that vector graphics are performance hogs. qb2DebugDrawSettings is a class containing only static variables that globally effect how and what is rendered by any drawDebug() call.

If you provide your qb2World with a debugDrawContext (a Graphics object), all objects residing in the world will automatically draw themselves to it using drawDebug(). This behavior can be toggled on the object level by setting qb2Object::drawsDebug to true or false.

Linked Lists vs. Arrays

QuickB2 internally uses arrays to store lists of data. This is a change from Box2D which uses linked lists exclusively. Linked lists have some speed advantages in certain situations, but there are three reasons arrays are used in QuickB2. First, I think arrays are typically easier to deal with, both for the developer (me) and a user of a library like this. Second, QuickB2 has some situations where z-order is important, and being able to easily change around z-order with array operations can be nice. Third (and this admittedly isn't a *great* reason), arrays are what most ActionScript 3.0 developers are used to what with manipulating display lists all the time; accordingly, QuickB2 uses similar conventions for its array manipulation functions (getObjectAt(), addObjectAt(), setObjectIndex(), etc.) to the DisplayObjectContainer class (getChildAt(), addChildAt(), setChildIndex(), etc.).

Units

Box2D is based in meters/kilograms/seconds/radians. QuickB2 is based in **pixels**/kilograms/seconds/radians, unless otherwise noted (e.g. forces and velocities use meters). The conversion is done for you through the 'pixelsPerMeter' property of qb2World. The default of 30 seems to be a good value for most simulations. The choice to use pixels is due to the fact that 2d programmers *ultimately* have to work in pixels anyway, so why make life complicated. If you really want to work in meters in your own code, simply set pixelsPerMeter to 1.

You should set the pixelsPerMeter property of qb2World once at the beginning of your program, and then never, ever touch it again.