ALTERED SOFTWORKS

Box2Scene Format Specifications

Version 1.0

Paril

1/2/2011

This document outlines the full specifications and standard implementation requirements of the Box2Scene XML format.

Contents

Introduction 2

Types 2

Layout 3

Shapes 4

Fixtures 6

Bodies 8

Joints 11

Introduction

Box2Scene is an XML-based format for describing an entire Box2D world. This includes bodies, fixtures, shapes, and joints. This article will guide you to the complete specifications of the format, including types, keywords, and anything else deemed necessary to implement the format in your Box2D port.

Types

Several different types are discussed throughout this specifications article. Here are their specifications:

- int
  - 32-bit integer (format: “5”)
- float
  - 32-bit floating point value (format: “6.4”, “8”)
- vector2
  - A simple element containing two floats. Two formats must be supported by your implementation: space-separated and comma-separated values (format: “1 5”, “1.0 7.4”, “1, 5”, “1.0, 7.4”). **Note:** Empty vector elements must be interpreted by the parser as an empty vector (0, 0). **Note:** Vectors do not follow any directional scheme; for example, the Y axis might be +up –down in one program, but –up +down in another.
- container
  - Not a type in itself, but means that this element is an element described elsewhere in the documentation. Containers used in this documentation also provide ordering for certain elements (first in a container is ID 0, second is ID 1, etc).
- ID
  - Indicates that this field must be an ID to a container. This is of special mention because nodes can be in any order, therefore an identifier must be given to each object before storing can be done properly.
- string
  - A string.
- type
  - A custom type that is described elsewhere in this document.
- bool
  - A Boolean. This may be formatted as either the string-form (“true”, “false”) or the integral form (“1”, “0”) respectively.

Layout

A Box2Scene format only contains one top-level node: World. Inside that node lay the rest of the contents of a scene.

The following attributes are valid inside of a World node (bold indicates mandatory attribute):

- **Version (int)**
  - **The scene file version. Version 1 is Box2D 2.1.2, Version 2 is Box2D 2.2.0.**

The following elements are valid inside of a World node (bold indicates mandatory attribute):

- Gravity (vector2)
  - The gravity vector for this world. **Implementation Note:** There is no default gravity. It is recommended to always specify your own gravity vector.
- **Shapes (container: see 'Shapes')**
  - **The container of Shapes.**
- **Fixtures (container: see 'Fixtures')**
  - **The container of Fixtures.**
- **Bodies (container: see 'Bodies')**
  - **The container of Bodies.**
- **Joints (container: see 'Joints')**
  - **The container of Joints.**

Note: The scene elements can be in any order. The parser must be able to interpret the file the same way, even if the four container types are in any order (ie, Joints moved to first, Bodies moved to second).

Example World layout (an empty world):

```
<World
  Version="1">
  <Shapes />
  <Fixtures />
  <Bodies />
  <Joints />
</World>
```

Shapes

A shape is, quite simply, a geometric shape that is used for collision purposes. Shapes can be re-used. The Shapes container must be formatted as follows:

```
<Shapes>
[shapes go here]
</Shapes>
```

OR

```
<Shapes />
```

in the case that the scene has no shapes.

Inside the Shapes container, there is only one valid element:

- Shape
    - A shape definition.

The following attributes are valid inside of a Shape node (bold indicates mandatory attribute):

- Type (string)
    - The type of shape that will follow. This is an enumeration in string format. The following values are allowed:
        - Circle
        - Polygon

The following elements are valid inside of a Shape node whose Type = “Circle” (bold indicates mandatory element):

- **Radius (float)**
    - **The circle’s radius.**
- **Position (vector2)**
    - **The circle’s central position. 0, 0 is the center of the circle.**

An example of a Circle shape:

```
<Shape
  Type="Circle">
  <Radius>0.5</Radius>
  <Position>0 0</Position>
</Shape>
```

The following elements are valid inside of a Shape node whose Type = “Polygon” (bold indicates mandatory element):

- **Vertices (container)**
    - **A container for nodes, each named “Vertex”, which contain a vector2 inside of them. See below for example.**
- Name (string)

    - An identifier that can be used to identify fixtures from the using application. Because of its' usage-specific nature, there is no specification on how to use or the format of this field.
- Centroid (vector2)
    - The centroid of the polygon. Using this element allows you to specify a custom centroid. **Note:** By default, Box2D will generate a centroid for set vertices. Make sure that your implementation sets the centroid to this value (if it exists: DO NOT assume 0, 0 if the element is not in the shape!) AFTER the vertices have been assigned.

An example of a Polygon shape:

```
<Shape
  Type="Polygon">
  <Vertices>
    <Vertex>-40 0</Vertex>
    <Vertex>40 0</Vertex>
  </Vertices>
  <Centroid>0 0</Centroid>
</Shape>
```

Fixtures

A fixture is the definition of a specific shape attached to a body. Fixtures can be re-used. The Fixtures container must be formatted as follows:

```
  <Fixtures>
[fixtures go here]
  </Fixtures>
```

OR

```
  <Fixtures />
```

in the case that the scene has no fixtures.

Inside the Fixtures container, there is only one valid element:

- Fixture
    - A fixture definition.

The following elements are valid inside of a Fixture node (bold indicates mandatory):

- Name (string)
    - An identifier that can be used to identify fixtures from the using application. Because of its' usage-specific nature, there is no specification on how to use or the format of this field.
- **Shape (ID)**
    - **The container ID of the shape that this fixture will utilize. A fixture with no shape is malformed.**
- Density (float)
    - The density of this fixture. Zero indicates a static object. **Note:** A body is malformed if it is a dynamic body that contains a fixture whose density is zero.
- FilterData (type: FilterData)
    - The filtering data of this fixture.
- Friction (float)
    - The friction of this fixture.
- IsSensor (bool)
    - Denotes if this fixture is a sensor shape (does not actually create contact points with other objects, but still collides).
- Restitution (float)
    - The restitution of this fixture.
- UserData (type: object)
    - Application-specific, language-specific and usage-specific user-described data. Because of its' usage-specific nature, there is no specification on how to use or the format of this field.

The FilterData type is a simple type container which contains three integral values in it. The following elements are valid inside of a FilterData node:

- CategoryBits
  - The category bit this fixture belongs to. **Implementation Note:** This value may be provided in either decimal or hexadecimal format.
- MaskBits
  - The categories that this fixture will collide with. **Implementation Note:** This value may be provided in either decimal or hexadecimal format.
- GroupIndex
  - The group index of this fixture. This allows a certain group of objects to either never collide (a negative value) or always collide (a positive value). **Note:** Groups always win over category/mask bits.

An example of a well-formed Fixture (assuming there is at least one shape in the Shapes container) as well as a well-formed FilterData (this is the default filtering for all fixtures who aren't given a filterdata):

```
<Fixture>
  <Shape>0</Shape>
  <Density>5</Density>
  <FilterData>
    <CategoryBits>1</CategoryBits>
    <MaskBits>65535</MaskBits>
    <GroupIndex>0</GroupIndex>
  </FilterData>
  <Friction>0.2</Friction>
  <Restitution>0</Restitution>
</Fixture>
```

Bodies

Bodies are the actual definitions of the objects that will appear in the world. They are created out of fixtures. Bodies cannot be re-used in any way, shape or form. The Bodies container must be formatted as follows:

```
  <Bodies>
[bodies go here]
  </Bodies>
```

OR

```
  <Bodies />
```

in the case that the scene has no bodies.

Inside the Bodies container, there is only one valid element:

- Body
    - A body definition.

The following attributes are valid inside of a Body node (bold indicates mandatory):

- **Type (string)**
    - **The body's type. This is an in enumeration string format. The following values are allowed:**
        - **Static**
        - **Dynamic**
        - **Kinematic**

The following elements are valid inside of a Fixture node (bold indicates mandatory):

- Name (string)
    - An identifier that can be used to identify bodies from the using application. Because of its' usage-specific nature, there is no specification on how to use or the format of this field.
- Active (bool)
    - Whether or not the body will be active in the world. Inactive bodies are, simply speaking, invisible, but still exist and can be turned on at any time.
- AllowSleep (bool)
    - Whether or not the body will be allowed to sleep.
- Angle (float)
    - The angle, in radians, of the body.
- AngularDamping (float)
    - The body's resistance to angular changes
- AngularVelocity (float)
    - The body's angular velocity (turning speed, basically).
- Awake (bool)

    - Whether or not the body will begin awake. Setting this to false will cause the body to hang where it is until it is touched by another body. It is recommended to set this to true if an object is lying on the ground.
- Bullet (bool)
    - Enabling this will make the body be under continuous collision detection at all times.
- FixedRotation (bool)
    - Whether or not this body can rotate.
- MassData (type: MassData)
    - Allows you to override the automatic mass data of the shape. If this element is missing, the body will use the mass calculated from its fixtures. **Implementation Note:** Sub-elements missing should be replaced by the actual calculated data. This way, if they only wanted to modify mass, they need not calculate the center and inertia manually and place them in the file as well.
- InertiaScale (float)
    - ???
- LinearVelocity (vector2)
    - The body's velocity.
- Position (vector2)
    - The body's position.
- UserData (type: object)
    - Application-specific, language-specific and usage-specific user-described data. Because of its' usage-specific nature, there is no specification on how to use or the format of this field.
- **Fixtures (container of IDs)**
    - **A container whose elements are named "ID" and have a value of the Fixture ID which will be added to this body. *Implementation Note: While a body with no fixtures is indeed malformed, it is allowed in this format, as fixtures may be added to bodies before the body is actually added in the application.***

The MassData type is a simple type which contains elements describing the mass, center of gravity, and rotational inertia of a body. It is used to override the existing mass that would be calculated by the body. The following elements are valid inside of a MassData node:

- Mass
    - The mass, usually in kg, of the body.
- Center
    - The center of gravity
- Inertia
    - The rotational inertia of the shape.

An example of a well-formed body (assuming that Fixtures container contains at least 2 fixtures) as well as an example of a MassData override:

```
<Body
  Type="Dynamic">
  <AngularVelocity>100</AngularVelocity>
  <Awake>True</Awake>
```

```
  <LinearVelocity>-800 0</LinearVelocity>
  <Position>0 20</Position>
  <MassData>
    <Mass>5.0</Mass>
    <Center>0, 0</Center>
    <Inertia>1.2</Inertia>
  </MassData>
  <Fixtures>
    <ID>0</ID>
    <ID>1</ID>
  </Fixtures>
</Body>
```

Joints

Joints are objects that can connect bodies in many different ways. Joints cannot be re-used in any way, shape or form. The Joints container must be formatted as follows:

```
<Joints>
[joints go here]
</Joints>
```

OR

```
<Joints />
```

in the case that the scene has no Joints.

Inside the Joints container, there is only one valid element:

- Joint
    - A joint definition.

The following attributes are valid inside of a Joint node (bold indicates mandatory):

- **Type (string)**
    - **The joint type. This is an in enumeration string format. The following values are allowed:**
        - **Revolute**
        - **Prismatic**
        - **Distance**
        - **Pulley**
        - **Line**
        - **Weld**
        - **Friction**
    - ***Implementation Note: Gear joints are not supported yet.***

The following elements are valid inside of a Joint node of any type (bold indicates mandatory):

- Name (string)
    - An identifier that can be used to identify joints from the using application. Because of its' usage-specific nature, there is no specification on how to use or the format of this field.
- UserData (type: object)
    - Application-specific, language-specific and usage-specific user-described data. Because of its' usage-specific nature, there is no specification on how to use or the format of this field.
- **BodyA (ID)**
    - **The first body in the joint. This is an ID of a Body in the Bodies container.**
- **BodyB (ID)**
    - **The second body in the joint. This is an ID of a Body in the Bodies container.**

- CollideConnected (bool)
    - Whether or not the connected bodies should collide with each other. This is false by default.

The following elements are valid inside of a Joint node of type “Revolute” (bold indicates mandatory):

- EnableLimit (bool)
    - Whether or not the limits are enabled.
- EnableMotor (bool)
    - Whether or not the motor is enabled.
- LocalAnchorA (vector2)
    - The anchor local to body A. If this element is not found, 0, 0 is the default.
- LocalAnchorB (vector2)
    - The anchor local to body B. If this element is not found, 0, 0 is the default.
- MaxMotorTorque (float)
    - The highest torque that can be achieved by the motor.
- MotorSpeed (float)
    - The speed the motor will spin at.
- LowerAngle (float)
    - The lower angle limit, in radians.
- UpperAngle (float)
    - The upper angle limit, in radians.
- ReferenceAngle (float)
    - The reference angle, in radians. This, basically, is added to the lower and upper limits.

The following elements are valid inside of a Joint node of type “Prismatic” (bold indicates mandatory):

- EnableLimit (bool)
    - Whether or not the limits are enabled.
- EnableMotor (bool)
    - Whether or not the motor is enabled.
- LocalAnchorA (vector2)
    - The anchor local to body A. If this element is not found, 0, 0 is the default.
- LocalAnchorB (vector2)
    - The anchor local to body B. If this element is not found, 0, 0 is the default.
- LocalAxisA (vector2)
    - The local translation axis in body 1.
- MaxMotorForce (float)
    - The highest force that can be achieved by the motor.
- MotorSpeed (float)
    - The speed the motor will spin at.
- LowerTranslation (float)
    - The lower translation limit, usually in meters.
- UpperTranslation (float)
    - The upper translation limit, usually in meters.
- ReferenceAngle (float)
    - The constrained angle between bodies.

The following elements are valid inside of a Joint node of type “Distance” (bold indicates mandatory):

- LocalAnchorA
  - The anchor local to body A.
- LocalAnchorB
  - The anchor local to body B.
- Length
  - The natural length between the anchor points.
- FrequencyHz
  - The mass-spring-damper frequency, in Hertz.
- DampingRatio
  - The damping ratio. Range is from 0 (no damping) to 1 (critical damping).

The following elements are valid inside of a Joint node of type “Pulley” (bold indicates mandatory):

- GroundAnchorA
  - The first ground anchor in world coordinates.
- GroundAnchorB
  - The second ground anchor in world coordinates.
- LocalAnchorA
  - The anchor local to body A.
- LocalAnchorB
  - The anchor local to body B.
- LengthA
  - The reference length for the segment attached to body A.
- LengthB
  - The reference length for the segment attached to body B.
- MaxLengthA
  - The maximum length for the segment attached to body A
- MaxLengthB
  - The maximum length for the segment attached to body B
- Ratio
  - The pulley ratio, used to simulate a block-and-tackle.

The following elements are valid inside of a Joint node of type “Line” (bold indicates mandatory):

- EnableLimit (bool)
  - Whether or not the limits are enabled.
- EnableMotor (bool)
  - Whether or not the motor is enabled.
- LocalAnchorA (vector2)
  - The anchor local to body A. If this element is not found, 0, 0 is the default.
- LocalAnchorB (vector2)
  - The anchor local to body B. If this element is not found, 0, 0 is the default.
- LocalAxisA (vector2)
  - The local translation axis in body 1.
- MaxMotorForce (float)

    - The highest force that can be achieved by the motor.
- MotorSpeed (float)
    - The speed the motor will spin at.
- LowerTranslation (float)
    - The lower translation limit, usually in meters.
- UpperTranslation (float)
    - The upper translation limit, usually in meters.

The following elements are valid inside of a Joint node of type “Weld” (bold indicates mandatory):

- LocalAnchorA
    - The anchor local to body A. If this element is not found, 0, 0 is the default.
- LocalAnchorB
    - The anchor local to body B. If this element is not found, 0, 0 is the default.

The following elements are valid inside of a Joint node of type “Friction” (bold indicates mandatory):

- LocalAnchorA
    - The anchor local to body A.
- LocalAnchorB
    - The anchor local to body B.
- MaxForce
    - The maximum friction force, in N.
- MaxTorque
    - The maximum friction torque, in N-m.

The following is an example well-formed Revolute joint definition (assuming that there are two bodies in the Bodies container: body 1 being a dynamic body above the static one, and body 2 being a static body):

```
<Joint
  Type="Revolute">
  <BodyA>1</BodyA>
  <BodyB>0</BodyB>
  <CollideConnected>True</CollideConnected>
  <EnableLimit>True</EnableLimit>
  <LocalAnchorA>0 12</LocalAnchorA>
  <LocalAnchorB>0 -8</LocalAnchorB>
  <LowerAngle>-0.7853982</LowerAngle>
  <MaxMotorTorque>10000</MaxMotorTorque>
  <MotorSpeed>3.141593</MotorSpeed>
  <UpperAngle>1.570796</UpperAngle>
</Joint>
```