

WonderBlocks Backend Design Notes

Joshua Dick and Gerard Dwan

9/19/2008

1 Background Information

This document details the design of the WonderBlocks backend, which executes on the Wonderland server. It should be noted that every Java class in the WonderBlocks backend follows the JavaBeans convention, and can be placed into one of three categories:

1. Classes representing objects that will be displayed in-world (Block, BlockConnection)
2. Classes that manage in-world objects (BlockManager, BlockConnectionManager)
3. Classes that assign IDs that managers can map in-world objects to (BlockID, BlockConnectionID)

2 Architecture

2.1 IDs

Each object that can be displayed in-world (Blocks and BlockConnections) has an associated ID number that is represented by a separate object (a BlockID or a BlockConnectionID.) This way, only one copy of each Block and BlockConnection object exists in memory, but they can be referenced multiple times by other objects via their IDs. It is the responsibility of the manager classes (BlockManager and BlockConnectionManager) to actually map IDs to the objects they represent. The ID classes statically store the next ID number to be assigned, so that each time a new ID object is instantiated, the ID number stored inside it is automatically incremented. Separate ID classes exist for each object type so that BlockIDs and BlockConnectionIDs can increment independently.

2.2 Blocks

The Block class represents a single WonderBlock. This class stores information such as WonderBlock shape, size, color, and attribute/value pairs (metadata.) Blocks can also be connected together, but they do not retain connection information directly; connections are actually represented by a separate BlockConnection class. To that end, each Block maintains a list of BlockConnectionIDs, where each BlockConnectionID is associated with the BlockConnection that includes that particular Block.

2.3 BlockConnections

A connection between two Blocks is represented by the BlockConnection class. There are special rules governing connections.

Namely:

1. No Block can be connected to itself.
2. No two Blocks can share more than one connection between them.
3. A Block can have any number of connections to other Blocks, provided that rule 2 is not broken.
4. A connection between any two Blocks can be directionless, directional, or bidirectional.

These rules are handled by the Block Connection Manager, which will be discussed later.

A BlockConnection object stores information about the two Blocks that share that connection, as well as the direction in which the Blocks are connected. An advantage of having distinct BlockConnection objects is that it is possible for connections to have metadata associated with them that is independent of the two Blocks sharing the connection. (For example, each connection could store a weight, which would allow one to create a weighted graph.)

It was mentioned before that each Block maintains a list of BlockConnectionIDs that are associated with connections involving that Block. In actuality, only one Block in a given connection actually keeps track of that connection's ID in its list of BlockConnectionIDs; the other Block stores no information about that particular connection and doesn't 'know' it is part of a connection. The BlockConnection class employs a loose parent/child analogy to distinguish between the two Blocks sharing the connection. For a given connection, the parent Block is the one that stores that connection's ID.

This analogy makes the most sense for directional connections; a parent Block always points to a child Block, so the parent Block keeps track of that connection's ID. The analogy works similarly but is less logical for directionless and bidirectional connections. Although there isn't a parent/child relationship between blocks that are connected bidirectionally or with no direction, the connection is still represented as a parent/child relationship internally. Regardless of the connection's direction, only the parent Block 'knows' that it is part of a connection.

See Figure 1 on the following page for a diagram depicting the relationships between Blocks and BlockConnections that were just described.

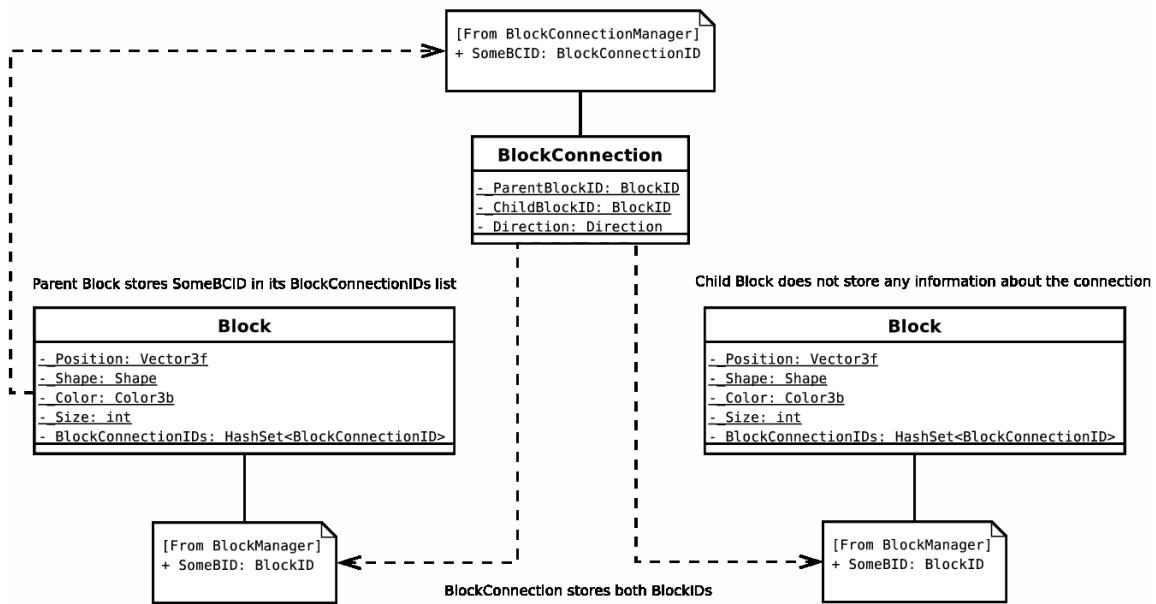


Figure 1: A not-quite-UML diagram depicting how a connection between two Blocks is represented.

2.4 The Manager Classes

The most complex classes in the WonderBlocks backend are the two manager classes, `BlockManager` and `BlockConnectionManager`. To facilitate easy sharing of the backend's state between its component classes, the WonderBlocks manager classes are singletons. They should be invoked accordingly by calling `*Manager.get*Manager()`. Normally, the singleton design pattern mandates that singleton classes should have protected constructors, so that the classes can't be manually instantiated (which would cause data synchronization problems.) The WonderBlocks manager classes DO NOT have protected constructors. Instead, these classes have public constructors, which are necessary for them to follow the JavaBeans convention. Therefore, be very careful when working with the manager classes. They should NEVER be manually instantiated.

The manager classes are the backbone of the WonderBlocks backend. Each manager abstracts away `Block` or `BlockConnection` objects so that they can be referred to and manipulated by ID number anywhere in the code, any number of times, while always maintaining a single copy of a given `Block` or `BlockConnection` object in memory. See Figure 2 on the next page for a simplified code listing demonstrating how `Blocks`, `BlockConnections`, and their respective manager classes work together to represent the overall state of an instance of the WonderBlocks component. For more specific information regarding how the manager classes should be used and manipulated, please see the WonderBlocks API documentation.

```

Vector3f boxPosition = new Vector3f(0.0f, 0.1f, 0.2f);
Color3b boxColor = new Color3b(new java.awt.Color(0, 128, 255));

Vector3f conePosition = new Vector3f(0.3f, 0.4f, 0.5f);
Color3b coneColor = new Color3b(new java.awt.Color(255, 128, 0));

Block testBox = new Block(boxPosition, Block.Shape.BOX, boxColor, 5);
Block testCone = new Block(conePosition, Block.Shape.CONE, cone-
Color, 2);

BlockID testBoxID = BlockMan-
ager.getBlockManager().createNewBlock(testBox);
BlockID testConeID = BlockMan-
ager.getBlockManager().createNewBlock(testCone);

BlockConnectionID testConnectionID =
BlockConnectionManager.getBlockConnectionManager().createNewBlockConnection
(testBoxID, testConeID, BlockConnection.Direction.PARENT_TO_CHILD);

```

Figure 2: A simplified code listing demonstrating how Blocks, BlockConnections, and their respective manager classes work together.

3 Data Persistence

Wonderland is built on top of Sun’s Project Darkstar game framework, which includes a Berkeley DB database for persistent storage. Unfortunately, the Wonderland 0.4 server erases this database every time the server is restarted, so the WonderBlocks component uses a different approach for persistent storage. Because every WonderBlocks class follows the JavaBeans convention, each of them can be serialized and stored on disk. At a later time, they can then be read back from the disk and be deserialized back into Java objects.

Remember that the manager classes abstract away Blocks and BlockConnections, and that Blocks and BlockConnections themselves refer to each other internally via ID numbers that are assigned by the managers. What this means is that serializing the manager classes to disk stores the entire running state of an instance of the WonderBlocks component, without including any redundant information (multiple copies of the same object.) Upon deserializing the saved data, WonderBlocks execution can resume as if the application had never terminated.

Note that the two managers have to be serialized to and deserialized from disk in the same order. In our current implementation of the WonderBlocks backend, the two managers are [de]serialized to/from XML files using Java’s built-in XMLEncoder and XMLDecoder classes. See Figure 3 on the following page for a simplified code listing that demonstrates [de]serialization of the two manager classes to/from XML files. In the future, we will create more generic interfaces to allow for [de]serialization, so that developers using the WonderBlocks backend will have more options for implementing data persistence. In other words, rather than saving all data to XML files, a developer

may wish to [de]serialize data to/from a database, etc.

```
BlockManager bm = BlockManager.getBlockManager();
BlockConnectionManager bcm =
BlockConnectionManager.getBlockConnectionManager();

//Serialize the two managers to an XML file
XMLEncoder e = new XMLEncoder(new FileOutputStream("output.xml"));
e.writeObject(bm);
e.writeObject(bcm);
e.close();

//Empty the variables, just in case
bm = null;
bcm = null;

XMLDecoder f = new XMLDecoder(new FileInputStream("output.xml"));
//Static methods deserialize each class when passed an XMLDecoder
bm = BlockManager.readBlockManager(f);
bcm = BlockConnectionManager.readBlockConnectionManager(f); f.close();
```

Figure 3: A code listing demonstrating how the manager classes can be [de]serialized to/from XML.