Open Rails Design Overview

This is an attempt to relate the design and operation of Open Rails (OR). It was written from my notes immediately following an inspection of the OR source code. (In some cases, it's quite possible that my interpretation is in-error. In that case, I hope someone will straighten me out.)

My inspection and note-taking sessions were usually related to a specific topic (e.g., terrain). I have hopes that I will, in the course of time, cover all important aspects of OR. I will introduce perhaps unfamiliar vocabulary as I go.

Contents

Major Components

Open rails is organized into three major components – Launcher, Menu, and RunActivity. A few words on each is in order.

Launcher

Launcher.exe (Launcher/Program) checks to see if prerequisite software (i.e., libraries) is installed. If all is well, Launcher starts Menu.exe and terminates.

Menu

Menu.exe (Launcher/Program) presents a dialog that allows the user to select a route and activity. Upon completion, it starts RunActivity.exe, passing an argument that contains the full route path and filename of the activity to be run. Menu remains resident while RunActivity runs, and the Menu dialog will be presented after RunActivity terminates at the end of the operating session.

RunActivity

RunActivity.exe (RunActivity/_Main) is passed a string argument with the full file specification of the activity to be started.[1] Main (_Main.cs) calls Start with the filespec as an argument. Start instantiates Simulator, whose constructor:

1. Creates an object TRK (instantiated from class TRKFile), whose constructor verifies that the specified folder contains a .trk file for either MSTS or ORTS. It instantiates either Tr_RouteFile (MSTS) or ORTRKData (ORTS), whose methods will parse the file.
2. Creates a TDB object (instantiated from TDBFile), whose constructor verifies the track database and returns.
3. TSectionDat (instantiated from TSectionDatFile) constructor verifies track sections and adds them to the class. Also does TrackShapes.
4. Activity (instantiated from ACTFile) constructor creates Tr_Activity_File (instantiated from Tr_Activity_File), which parses the activity file.

Immediately after Simulator's constructor returns, Start calls Simulator.Start, which does the following:

1. Initializes any time recording required by the activity.
2. Aligns all switches to their default positions, as specified by the activity. The internal Track Data Base (TDB) data structure is traversed in order to do this.
3. Places the player train.
4. Places static consists.
5. Signals will be initialized here when implemented.
6. Creates a queue of AI trains.

When Simulator.Start returns, Start instantiates Viewer (from class Viewer3D). Viewer's constructor does the following:

1. Sets up user game settings.
2. Creates and initializes the sound engine.
3. Initializes the environment according to the activity.
4. Reads TTYPE.DAT and builds an internal data structure.
5. Instantiates a Tile class object whose constructor initializes an 8x8 (x, z) buffer of tile objects.
6. Creates subsidiary two or three subsidiary threads, one for rendering, one for loading, and (if the PC has multiple processors) one for updating. Three classes are instantiated to run in these threads: RenderProcess, LoaderProcess, and UpdaterProcess. (Their constructors run immediately upon instantiation.) More on this below.

Immediately following the return of Viewer's constructor, Start calls Viewer.Run which calls RenderProcess.Run. RenderProcess.Run runs in the RenderProcess thread, and it calls XNA Run ("game loop").

The multi-threading (main thread, RenderProcess, LoaderProcess, and UpdaterProcess) is an important element of Open Rails' design. It enables the operating system to exploit systems with multiple processors.[2]

Let's outline the roles of the three concurrent subsidiary threads.

[1] To run RunActivity in Debug mode under Visual C#, insert a file specification (in quotes)in project Properties/Debug/ Command line arguments, and launch the project with Debug.

[2] Fire up the Resource Monitor associated with Windows Task Manager when Open Rails runs in windowed mode to watch the multiprocessing fun.

Render Process

When RenderProcess is instantiated, its constructor builds a window and instantiates an XNA GraphicsDeviceManager class. As mentioned above, after the constructor returns, RenderProcess.Run is called, which starts the XNA game loop. This will run, synchronized with the refresh rate of the monitor, processing all events associated with the game window, until the game ends.

With the game loop running and the graphics device ready, but before the first frame is displayed, RenderProcess.Initialize will be automatically called. This performs the following:

1. Viewer and materials initialization.
2. Creates and initializes all renderers ("drawers") and cameras.
3. Loads terrain, scenery, and trains for the starting position of the player train.
4. Starts the LoaderProcess thread that will run a game loop responsible for loading "content" (terrain, scenery, trains, etc.).
5. Similarly, it starts the UpdaterProcess thread for a multiprocessor system.
6. Unpauses Simulator.

Paced at the monitor's native frame rate, RenderProcess.Draw gets called to actually render all the primitives for the next frame (prepared by UpdaterProcess).

Loader Process

Once LoaderProcess has started, it runs a loop that waits until signaled to start. Then it loads any new graphics content required by the camera position, and it signals when it is done and branches back, ready to do it again.

Updater Process

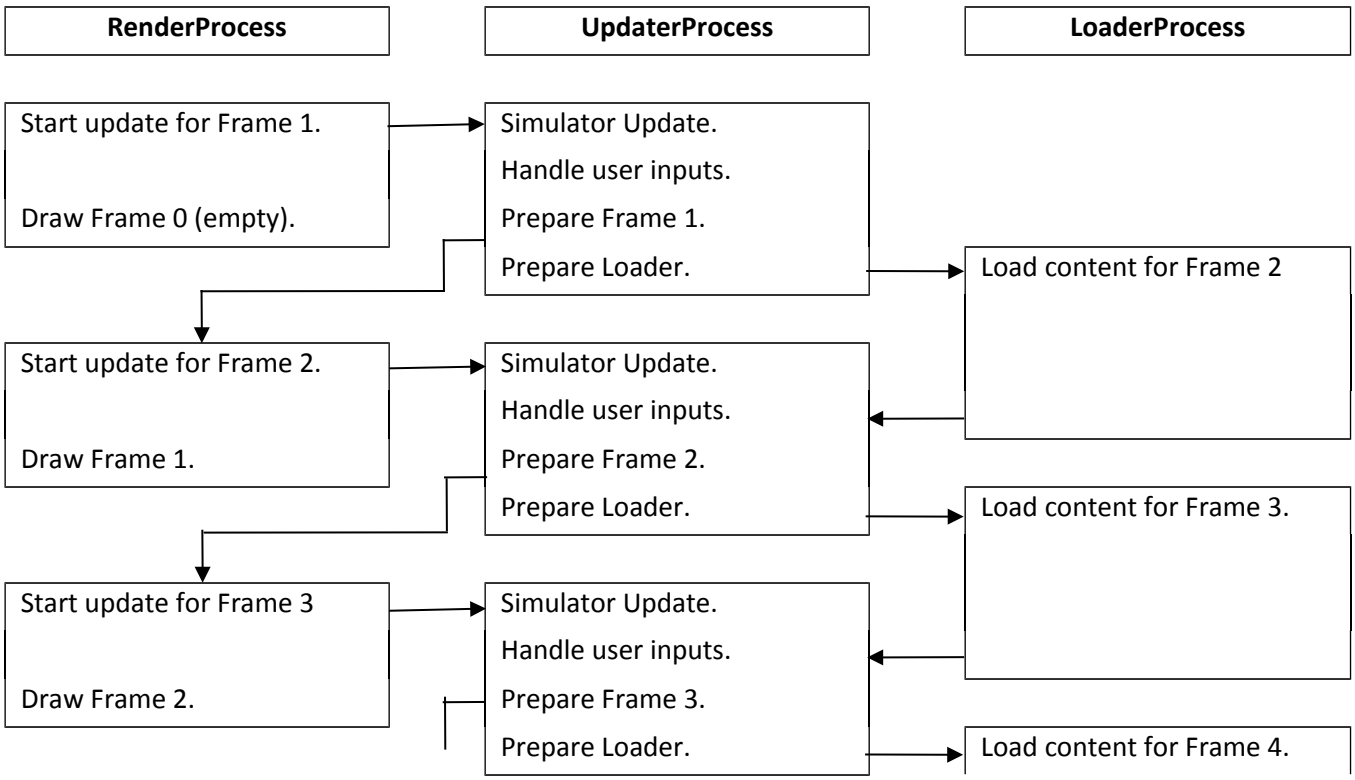
Let's assume (temporarily[3]) that our configuration has more than one processor.

UpdaterProcess also runs a loop (UpdaterProcess.UpdateLoop) that is responsible for preparing each frame for rendering. The loop begins with a wait until signaled to start. Then it does the following:

1. Keeps track of real time (wall clock) and computes frames-per-second (FPS).
2. Calls Viewer.Simulator.Update to update the state of the simulation.
3. Handles user inputs that were read in the RenderProcess thread.
4. Prepares a frame for display. This involves calling a number of specialized preparers (sky, terrain, scenery, etc.) that queue up "primitives" (e.g., mesh) for rendering. The list of preparers is:
    a. Camera.PrepareFrame
    b. SkyDrawer.PrepareFrame
    c. TerrainDrawer.PrepareFrame
    d. SceneryDrawer.PrepareFrame
    e. TrainDrawer.PrepareFrame
    f. PrecipDrawer.PrepareFrame
    g. InfoDisplay.PrepareFrame
5. Signals RenderProcess that it can proceed to render the frame just prepared.
6. Signals UpdaterProcess that it can proceed to load new content.

The behavior of the above three processes must be synchronized. The descriptions above hint at some of that synchronization. The simplified illustration below attempts to illustrate the time-wise coordination of the processes. (Time flows down the page. The time axis is, in no sense, to scale.)

[3] I presume that updating is done in the main thread in the uniprocessor case. However, I really don't know the details of how this is handled. (More on this as my orientation progresses.)



Open Rails Terrain

Terrain Fundamentals

If you've done any MSTS route-building or read forums in depth, you know that a route's "world" is divided into *tiles* 2048 kilometers square. Further, you'll recall that a tile is further sub-divided into a 16x16 grid of *patches*, each 128 meters square.

MSTS further divides each patch into a 16x16 mesh of *vertices*. Each set of four adjacent vertices defines a "cell" with two *triangles*. Hence, each patch has 2x15x15 = 450 triangles. Another name for a set of adjacent triangles is a *mesh*. Such a mesh is considered to be a graphics *primitive*.

In Open Rails, such a patch is represented by a *class* named TerrainPatch (derived from a more general (base) class, RenderPrimitive). When a terrain patch object is created (an *instance* of TerrainPatch), the class *constructor*, called at creation time, gets patch information from the respective TFile (created from a .t file) and the y-coordinate (elevation) of each vertex from the respective YFile (created from a _y.raw file). The TFile information is associated with the patch; the YFile information (elevations) is individually associated with the vertices. Hence, the YFile information displaces the vertices vertically, forming a shaped terrain patch. MSTS fixes the x- and z- coordinates in place and does not move them in those directions. This is not generally the case with patches and need not be the case with Open Rails.

During the very first execution of the TerrainPatch constructor, it calculates an index buffer, which is shared by all subsequently created TerrainPatch objects. This is possible because all TerrainPatch objects use the same size grid.

Associated with each patch vertex are three vectors: a *position vector* (x, y, z), a *normal vector* (N_x, N_y, N_z) approximating a direction perpendicular to the terrain surface at the vertex, and a *texture coordinate vector* (u, v), which is used to determine exactly how a *terrain texture* will be wrapped over the patch surface. The three vectors (position, normal, texture coordinate) are represented by an XNA VertexPositionNormalTexture structure. These three vectors have a total of 8 components (3 + 3+ 2), each component requiring a four-byte floating point number (*float*). Hence, each patch requires a minimum of 16x16x8x4 = 8192 bytes.

The terrain texture will come from MSTS folder TERRTEX. Terrain textures will be image files (.ace for MSTS) which may be of varying size, but typically 512x512.

With that groundwork established, let's proceed to examine how tiles are handled.

Tile Handling

Shortly after the simulator starts, Simulator.Start (a method) creates a Viewer3D object called Viewer (a class). Viewer's constructor does a bunch of things, but only one thing is related to terrain: It creates a class called Tiles (note the plural). The Tiles class is a simple one; it includes only:

1. The constructor Tiles.
2. A two-dimensional [tileX, tileZ] buffer, each element of which contains a *reference* to a tile. (Elements don't contain the tile's data; they contain a *reference* to the data.) The buffer, private to the class, is called TileBuffer. The size of the buffer is 8x8.
3. A method, GetTile, which returns a reference to a requested tile. GetTile maintains the buffer. If the tile requested by [tileX, tileZ] is not in the buffer, GetTile creates a new tile object (an instance of class Tile), reading the data comprising the tile from a file.
4. A method, GetElevation, which returns a terrain elevation from [x, z] within the tile referenced by [tileX, tileZ].

Each tile read and added to is represented as an instance of a Tile (singular) class. Each Tile class instance created includes a TFile class (by reading a .t file) and a YFile class (by reading a _y.raw file). The YFile constructor reads a 256x256 array of Uint16s (_y.raw file) representing a scaled elevation value between 0 and 65,535. (The RAW file is a scan of elevations from the NW corner to the SE corner in row-major order.)

The Tile class is pretty simple. It includes:

1. Tile, the constructor
2. A TFile class.
3. A YFile class.
4. A, a 256x256 matrix containing the elevation values.
5. IsEmpty, a Boolean flagging whether the tile has been read.
6. A method, GetElevation that fetches an elevation value given interior coordinates [x,z].

TDBTraveller

A TDBTraveller is responsible for traversing the route (as defined by the track database). A reasonable model for a TDBTraveller is a truck that follows a path on the route. It specifies the position and direction (orientation) of the truck. Each instance of class Train (including AI trains) has two references to instances of class TDBTravellor – FrontTDBTraveller and RearTDBTraveller – which represent the position and direction of the front and rear of the train.

When Simulator.Start calls private method InitializePlayerTrain, it (InitializePlayerTrain) reads the service and consist files and instantiates a Train object. Next, it instantiates a PATTraveller, which identifies the position of the rear of the train in the track database (TDB). A new TDBTraveller is created with this initial position, and a reference to it is stored in train.RearTDBTraveller. InitializePlayerTrain then proceeds to build the consist. With the consist completed, InitializePlayerTrain calls train.CalculatePositionOfCars. CalculatePositionOfCars creates a new TDBTraveller named traveler which is a copy of RearTDBTraveller. It then steps traveller forward, a truck (bogie) at a time until it reaches the front of the train. Then, a reference to traveller, now at the front of the train, is assigned to FrontTDBTraveller. The rear-to-front traversal is done in three steps per car (or engine):

1. The traveller is moved from the rear of the car to the position of the rear truck.[4]
2. The traveller is moved forward to the position of the front truck.

[4] Truck-to-truck spacing is approximated as 65 percent of car length because the spacing is not reported in the .wag file.

3. The traveller is moved forward to the front of the car.

The step-wise traversal is done by traveller.Move.

The positions of the traveller at each of the trucks are used to derive a transformation matrix that correctly orients the car and positions its center.

Appendix A – An Object-Oriented Programming Primer

The following description of Object-Oriented Programming (OOP) terminology is offered in the context of the C# programming language.

Class – A class represents an object in OOP parlance. A class can contain a number of types of *members*, most importantly, *structures* and *methods*.

Constructor – A method that is called automatically when an *instance* of a class is created. Commonly, used to initialize members of the class.

Derived From – The Apple class and Orange class are derived from the Fruit class.

Instance – One occurrence of a class. For example, each apple is an instance of the Apple class.

Method – A member of a class that takes arguments and returns a result. In traditional terminology, a function.

Reference – Think of it as a pointer to an item (e.g., data, class).

Structure – A member of a class that structures one or more data items.

Appendix B – Keyboard Controls

F2 – Save.

F5 – HUD1/HUD1+HUD2/none.