A-OK!/RB3D Demo

A Demo Package for the **RB3D** Plugin and **A-OK! THE WINGS OF MERCURY**

Description

A-OK!/RB3D Demo is a REALbasic source program (version 3.0 format) that demonstrates some basic and no-so-basic techniques for using Joe Strout's RB3D real-time 3D engine plugin for REALbasic. RB3D allows you to create interactive real-time 3D worlds using an extension of the REALbasic syntax.

A-OK! The Wings of Mercury is a highly detailed, technically and historically accurate simulation of the Mercury spacecraft. An older version was developed and marketed by myself in the mid-90's for the Mac OS. With the commercial introduction of REALbasic in 1997, I decided to completely re-write the program in REALbasic, adding additional realism like fully rendered control panels, full fidelity trajectory simulation, astronaut physiology simulation, networked mission control console and real time 3D graphics. The last requirement caused me great grief. After trying several methods including a "commercial" plugin and some wrapper classes that allow programming directly in OpenGL and QuickDraw 3D (and many re-writes in the process), I settled on RB3D. Joe Strout has been an invaluable help to me since the beginning, offering his knowledge of real-time 3D programming throughout development.

This demo includes some core code and some of the current 3D models from the commercial version of A-OK! The Wings of Mercury 3.0. The main panel is just a hack for demo purposes; the actual product is eons more realistic. Also, there are many operation details that are not present in this demo. The point here is that this is a demo of RB3D, not A-OK!

Software Requirements

In addition to REALbasic 3.0, the demo requires the very latest (2-OCT-2000) release of the RB3D plugin, OpenGL 1.1.2 or later, QuickDraw 3D 1.6. There are links to each site where this software can be found in the main folder. The compiled version runs on Mac OS 8.6 or 9.0.4.

Hardware Requirements

At least a 200 MHz PowerMac is required. An 8Mb 3D Accelerator card that supports OpenGL is also required.

Program Operation

Upon startup, the Booster Cutoff Parameter window appears. You can just click Done and you will get a standard Mercury orbit. If you change the parameters, you'll get a different orbit. Try increasing the cutoff velocity and you'll get a very high apogee (apogee - highest point in an orbit, perigee - lowest point). If the Earth disappears, you have created an orbit that intersects the Earth - you're dead.

Because the Earth model is huge, it'll take about 30 seconds for the windows to appear on a 200 MHz machine. There are two windows, the main Capsule window and the External View window. Initially, only the Capsule Window is visible. To open the External View window, select External View from the File menu or type command-E.

The External View window follows the spacecraft around the Earth. You can view the spacecraft from 6 different viewpoints and vary the distance from the spacecraft. The Reset button resets the distance in case you've overdone it a bit.

The Capsule window has a "picture window" view. The actual Mercury spacecraft had a much smaller, vertically oriented window and the A-OK! product models that. There are text gauges that show attitude rates and position, the current altitude and the apogee and perigee of the orbit created. Again, these text gauges are just for the demo; the product models the real gauges. Also, no attempt was made to draw off screen, so there is some flicker in the gauges.

When the Capsule window is front most, the arrow keys are used to fire the attitude thrusters. Up/Down pitch up and down. Left/Right yaw left and right. Shift Left and Shift Right roll the spacecraft left and right. If you get out of control the Rate Damping button on the upper right activates a simplified version of Mercury's Rate Damping system, firing thrusters until all rates are zero. In this demo, no fuel is consumed; the product and real Mercury consumed fuel furiously if attitude control was sloppy and the first two missions came back with dry tanks!

The other two buttons on the upper right control the jettisoning of the retrorocket package and the antenna canister. Note that in the actual mission the antenna canister would not be jettisoned in orbit as it contains items useful for landing: parachutes! Also the retro package would only be jettisoned after retrofire. In this demo you can see the jettisoning action by pressing these buttons. If you jettison the retro pack, yaw 180° to watch it from the Capsule window. You can see the tip of the antenna canister at the bottom of the Capsule window. When it is jettisoned, it start falling "down", so you'll need to pitch down to follow it in the Capsule window.

The buttons on the lower right allow you to fast forward and pause the simulation. Finally, the Reset button closes the view windows and opens the Booster Cutoff Parameter window allowing you to enter new parameters.

Program Structure

capsuleWindowPanel is the main window of the program which holds the capsule view, all text gauges, all simulator and spacecraft controls, the key processing code and the update timer. Things to remember here are that the timer runs at 0.1 of a second normally and 0.01 of a second in fast forward mode. Also, the keydown method simulates the firing of thrusters by adding or subtracting a value to the appropriate axis rate variable. Note that this window must be the top most window to process the key commands.

initCode is the application module that really just holds the call to the startup method upon open and clears all 3D objects out before the RB3D views close. This is a quirk in the current build of RB3D: if you don't nil out all 3D-object references before the window containing View3D are closed, you crash.

cutoffWin is opened when the program first starts up or is reset. It allows you to enter the altitude that you entered orbit and the final cutoff velocity of the booster. Default values always are entered, so you can just hit done if you'd like a default orbit.

externalViewWindow houses another RB3D View3D object and some buttons to control the view position and distance of the external camera to the spacecraft.

aokTrajectoryGlobals contain the code for calculating the orbit based on booster cutoff parameters, a simple method for updating the attitude based on the attitude rates, a *rateDamping* method that fires the thrusters to counteract all angular momentum until there is no more angular movement, a method for updating the spacecraft's orbital position.

aok3dObject is a class that is the basic 3D-object class for A-OK! 3D models. (as if the name didn't make that obvious!) The class contains all the parameters over and above the basic ones provided by RB3D's Object3D class on which it is based. It also contains an update method that is used to update the object's position and orientation once it is jettisoned from the spacecraft.

aok3dGlobals is the main suite of A-OK! specific 3D methods, constants and properties. Included are a method to change the external view; a method to initialize all the models; a method to load a model from a 3DMF file; a method to handle the jettisoning of a spacecraft part; a method to update the position and orientation of a part while it is attached to the spacecraft and *update3dObject*, the method that updates the position and orientation of all the spacecraft parts, both attached and jettisoned, the Earth and both the capsule and external View3D class.

Trajectory Engine

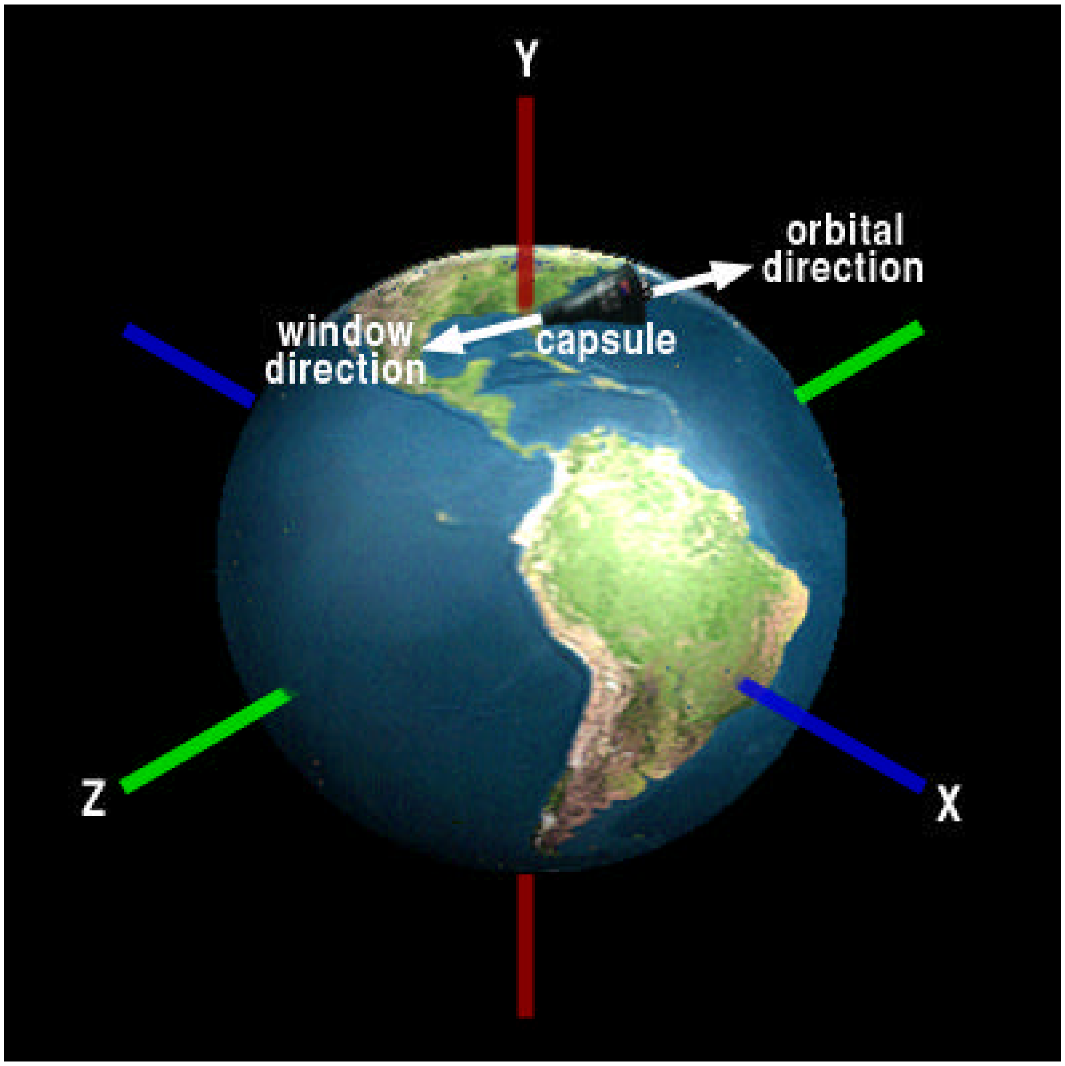
The Mercury spacecraft was placed into a 104 by 211 statute mile orbit by an Atlas D rocket. The spacecraft was launched from Cape Canaveral Air Station or what has become the Kennedy Space Center. Simulating orbits usually requires calculations utilizing all three dimensions. Because A-OK! only simulates Mercury and because Mercury was not able to change orbital parameters (except to reenter and land), we can use a simpler method and a little trick.

The *updateOrbit* method simply calculates the path along a 2D elliptical path. If we did nothing else, we would be flying over the equator, inviting but inaccurate. Therefore, when the Earth model is loaded by the *init3dEnvironment* method, it is then rotated so that the starting point is right over the Cape and the path is angled to simulate a 32° orbital inclination. This way, we only calculate Y and Z coordinates; X always remains zero.

The *calculateOrbit* method creates all the parameters needed to define the 2D elliptical path, based on the cutoff altitude and velocity obtained from the *cutoffWin* window.

The diagram below shows the unusual coordinate system this creates. It also shows another Project Mercury quirk. Because mission planners wanted the spacecraft to be ready for retrofire and reentry at any time, the spacecraft flew with it's back facing the direction of orbital travel. In other words, butt first! This means that the normal view is of scenery you just passed!

The trajectory engine also handles the attitude of the spacecraft. The spacecraft's initial orientation is 0° pitch, 0° yaw and 0° roll which is defined, *in the trajectory engine's coordinate system,* as window up and front opposite of direction of travel. Remember that "up" is relative to space not the Earth. If its attitude does not change as it orbits the Earth, the spacecraft's window will be facing down, with respect to the Earth, in about 45 minutes. The actual product (and actual spacecraft) is able to follow LHLV (local horizon, local vertical) coordinates in which "up" is always away from the Earth. This was done using horizon scanners and the attitude thrusters to track the local horizon. Finally, this demo does not model the gyroscope package, which drives the attitude displays in the real spacecraft. The demo's gauges never drift.



3D Engine

The models were constructed using Meshwork, a 3D modeling program specifically designed for creating low-polygon models for use in real time 3D engines. The textures were created either as original artwork or from massively manipulating public domain photographs from NASA.

The *init3dEnvironment* method loads all the models into memory. Aside from the odd orientation, the Earth model is straightforward and is the center of the coordinate system used by the trajectory engine.

Next, the three parts of the demo spacecraft are loaded into an array of *aok3dObject* classes (which is a subclass of RB3D's Object3D). The class contains: a vector which defines the offset of the part to the center of the spacecraft model, a quaternion that represents the current orientation of the spacecraft and therefore the orientation (and, to an extent, path direction) of the part once jettisoned, storage for X, Y and Z velocities and pitch, yaw and roll rates and finally, and integer that stores the lifetime of the jettisoned part. One method, update controls all aspects of the jettisoned object's movement.

In the actual product, there are currently over 40 parts to the spacecraft model, including booster parts, multi-shape flames, parachutes, thrusters, deployable experiments, etc., that stay within the spacecraft, until jettisoned. Obviously, some parts like flames and parachutes remain invisible until needed.

The *capsuleCabin* model is the "home" model around which all others are placed since it is the only piece of the spacecraft that returns to Earth. In this demo, there are two additional parts, the *capsuleRetro* and the *capsuleAntennaCanister*. Note that while these two parts are offset in the Y direction only (retro negative, canister positive), other parts (like the thruster flames) would require a full 3 axis offset. The problem with this is that I do not have separate standalone editor that would allow me to place parts visually, then generate the proper offsets. Instead, I created most parts in Meshwork with the cabin model in place, after which, I removed the guide model. All I had to do in code was mess with Y offset values which was fairly easy to do.

All models were created in the default Meshwork coordinate system. This corresponded to the orientation for launch. Since the product allows you to start in orbit, the next section of *init3dEnvironment* puts the completed model through a set of rotations that match the gyrations the spacecraft goes through from launch to orbital insertion.

Okay, so the trajectory engine gives us a set of coordinates that we simply pass on to the 3D engine. Rather simple, right? In many cases this would work, but A-OK! is not your normal first person shooter. Because the trajectory numbers were based on reality, my initial attempt was to make the 3D environment full scale. Yes, one meter corresponds to one unit in the 3D world. There were two fundamental problems with this.

One, if we put the center of the 3D engine coordinate system as the center of the Earth that means our object of interest, the spacecraft is the radius of the Earth, plus the altitude of the orbit away from the center or 6378150 + 163000 (about 100 miles) = 6541150 units!!! The problem is that the hither and yon values of the View3D class are based on integers. In order to put the yon plane far enough away to allow the capsule view camera to see the entire Earth, I had to set the value close to the limit of an integer. This produces a visual mess as the image was very jerky and sections of it fell apart with the slightest change.

The answer was reducing the size of the 3D world by one tenth. In the 3D engine I apply a scaling factor to all vectors passed from the trajectory engine, so I can work in full scale in the simulator code. This is important because the product version actually records and graphs all data, so it has to be full scale. This improved the view.

Before we go on to the next problem, we should review the camera placement strategy. Two View3D classes act as cameras, one for the *capsule3dView.camera* and one for the *external3dView.camera.* In *init3dEnvironment*, the *capsule3dView.camera* is rotated and offset in the Z-axis to simulate the astronaut's eye position. This camera "moves" with the spacecraft and matches its orientation. The *changeExternalViewCameraMode* handles the placement of the external camera which also "moves" with the spacecraft, always pointing to the center.

Now the final problem. I originally put all the spacecraft objects into an RB3D Group3D. It made sense since I would only have to translate and rotate one object. Except in practice, it the objects started to fall apart as the spacecraft moved. It was very disturbing to watch an entire launch vehicle come apart as it soared to the heavens. I imagine that this was not too dissimilar to what the original Mercury astronauts must have felt when they watched a test launch of the Atlas booster—the one they were supposed to ride on top of—blow up! Joe Strout warned that this was not the best method in his original RB3D documentation and that was proving true (although for some scenarios it would work).

Then I created the array of objects, *spacecraft3dObjects()* and a method, *updateSpacecraftGroup* to update the position and orientation of each part, based on the orientation and position of the entire spacecraft. I step through *spacecraft3dObjects()* calling this method. This worked better, but still resulted in some slight, but unacceptable separation of parts during rotation.

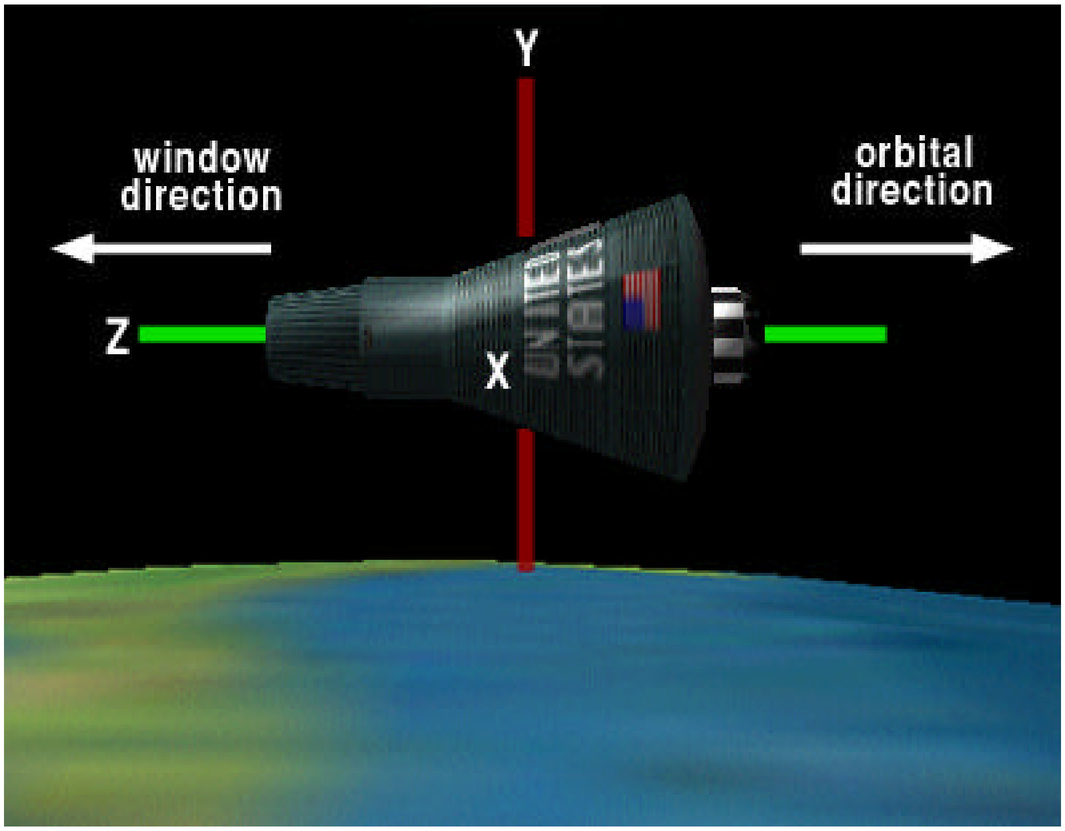
After a series of weeping emails from me, Joe Strout came up with a wild idea. The concept was the most confusing set of gyrations my brain has ever been put through**: keep the cameras still and move everything else!**

In the previous section, we described an odd coordinate system with the polar axis running directly through Florida. Under this new scheme, the 3D-engine coordinate system is centered on the spacecraft cabin. The diagram below shows this and should be compared to the one above.

For example, launching the spacecraft under the "normal" way translates the spacecraft in a positive direction along the Y-axis. Now, launching involves translating the Earth along the Y-axis in a negative direction! Okay, but how do I keep track of the spacecraft position and overall orientation? Before, I just translated and rotated the spacecraft Group3D. I created another *aok3dObject*, *spacecraft*, for the purpose of having a place to store the current position and orientation of the entire spacecraft.

The *update3dObject* method is the main code that does all this. It takes the vehicle's X, Y and Z coordinates and the pitch, yaw and roll rates supplied by the trajectory engine methods does the following:

1. Update the Earth *aok3dObject* (and in the product other "fixed" objects) position, scaling by 0.1
2. Update the *spacecraft aok3dObject's* position, scaling by 0.1
3. Update the *spacecraft aok3dObject's* orientation
4. Step through each spacecraft part contained in *spacecraft3dObjects()*
    · If not jettisoned, *updateSpacecraftGroup* applies a transformation to the part so that it's position and orientation stays in sync with the *capsuleCabin aok3dObject*, the "home" object.
    · If jettisoned, (a flag in the *aok3dObject* class), the part's individual *update* method handles the part's path and orientation.
5. The *capsule3dView.camera* is transformed based on the *spacecraft aok3dObject* and the *capsuleCameraOffset*, which moves the *capsule3dView.camera* to match where the astronaut's eyes would be.
6. The *capsule3dView.camera* is then rotated using the same attitude rates.



Conclusion

I hope this demo and document helps the REALbasic and RB3D community. My thanks go out to Joe Strout for his advice, Meshwork and above all RB3D, without which A-OK! would be no more. Additional thanks go to the entire REALbasic team, past and present; A-OK! 3.0 would never have been attempted without REALbasic either!

Joe Nastasi, YOU ARE GO! Educational Space Simulations, jnastasi@u-r-go.com

Legal