cloudBloomer
workflow / manual /

---

3. Using cloudBloomer2

cloudBloomer is a short piece of C++ code that uses openFrameworks* to do some manipulation of pointcloud data. It works with one pointcloud file (the .txt file you created in textEdit from a .ply file). By setting various parameters in the code, you can geometrically manipulate the pointcloud, causing it to (for example) rotate or expand over time.
cloudBloomer produces output in two modes, 'live; and 'save'. In 'live' mode, it renders the manipulation in a window on screen; in 'save' mode, it writes rendered frames as a sequence of images to disk (that you can later import into DaVinci Resolve as a movie).

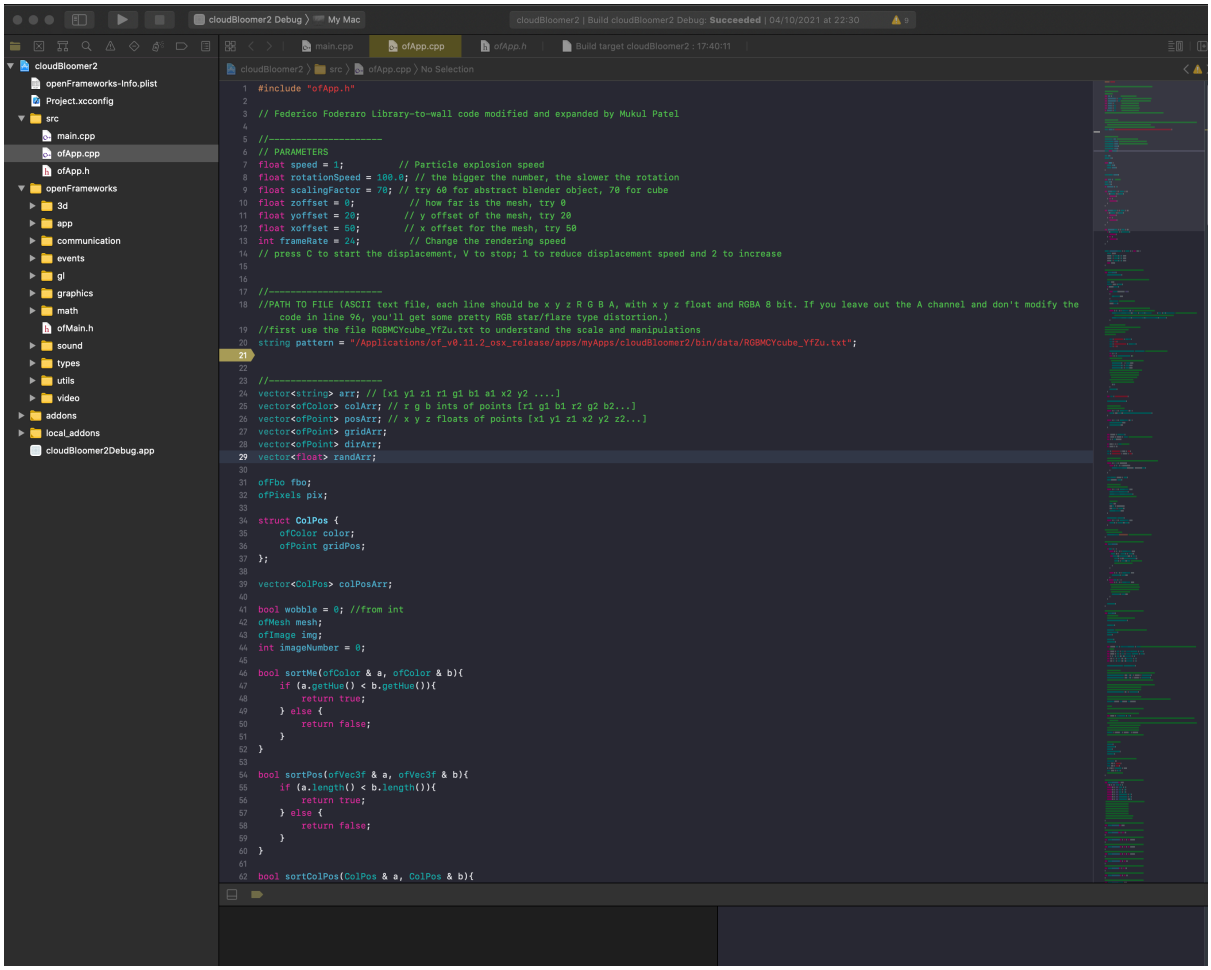
Summary workflow:
Open cloudBloomer as C++ code in Xcode (Apple's Integrated Development Environment or IDE, which is an application that helps you develop code).
Choose a pointcloud.
Adjust parameters in the code.
Check you're in 'live' mode.
Build and run the code, and observe the results.
Stop and adjust parameters as necessary, and rebuild.
When you've got something you want to capture, amend the code to 'save' mode so that it writes to disk, and build again.
Closing Xcode saves the code in the last-edited state.

IMPORTANT
You MUST preserve the indentation and punctuation when editing .cpp files – they are important parts of the syntax. But anything on a line after // is a comment and is ignored when the code is being compiled. Xcode will warn you of syntax errors.
Also, adding blank lines in the code will have no effect, except that it will change the numbering of subsequent lines, so to make communication in class simpler, avoid doing it if possible.

*Since the move to ARM silicon, Apple is slowly abandoning OpenGL (which is what we use openFrameworks to give us easy access to). The cloudBloomer code should still work for another year or two, but keep an eye on
https://openframeworks.cc
for updates.

3.1 Opening the cloudBloomer Xcode project

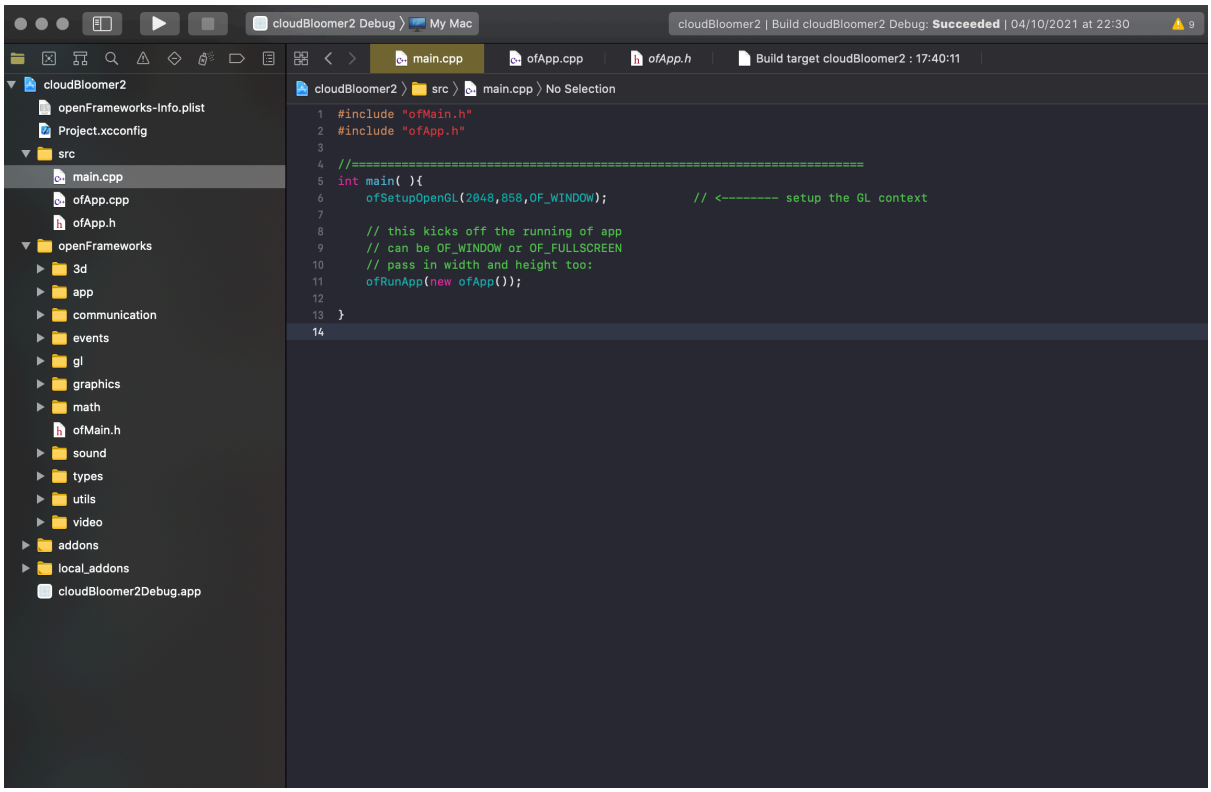
Find
cloudBloomer2.xcodeproj
and double-click to run. Xcode will open, and you'll see something like this.



The main part of the code is ofApp.cpp.

3.2 Set aspect ratio

From the left hand column, under src (source code), choose
main.cpp



and amend the window size in line 6 to match your target output size (2048 x 858 is 2K cinemascope; you might want to use 1920 x 1080 full HD instead – so the line would read:

```
ofSetupOpenGL(1920,1080,OF_WINDOW);          // <-------- setup the GL context
```

You won't need to change main.cpp again unless you want a different aspect ratio.

Now, from the left hand column, under src (source code), choose
ofApp.cpp
and amend the lines 148, 151, and 189 (this is for a target resolution of 1920 x 1080):

```
float xquadro = totPoints / 1.778f; // 2.387f for 2048x858 1.778f 1920x1080
```

```
int xwidth = x * 1.778;//aspect ratio
```

```
fbo.allocate(1920,1080,GL_RGB32F_ARB); //aspect ratio 570x1024 inf obj native
```

The figure 1.778 is the aspect ratio (1920/1080).

3.2 Choose a pointcloud

Amend line 20 so that it points to the file you want to open:

```
string pattern = "/Applications/of_v0.11.2_osx_release/apps/myApps/cloudBloomer2/bin/data/RGBMCYcube_YfZu.txt";
```

You can simply drag the file into this line of the code, and Xcode will automatically put the correct path to the file in. Just make sure that the path is in double quotation marks, and that the line ends with a semicolon. For orientation and reference, first work with

RGBMCYcube_YfZu.txt

This is a unit cube (edge length 1) with faces coloured red, green, blue, magenta, cyan and yellow, and oriented with the Z-axis up and the Y-axis forward (out of the screen, towards you). Once you know what the various manipulations do to this, you can better understand what they will do to more complex shapes.

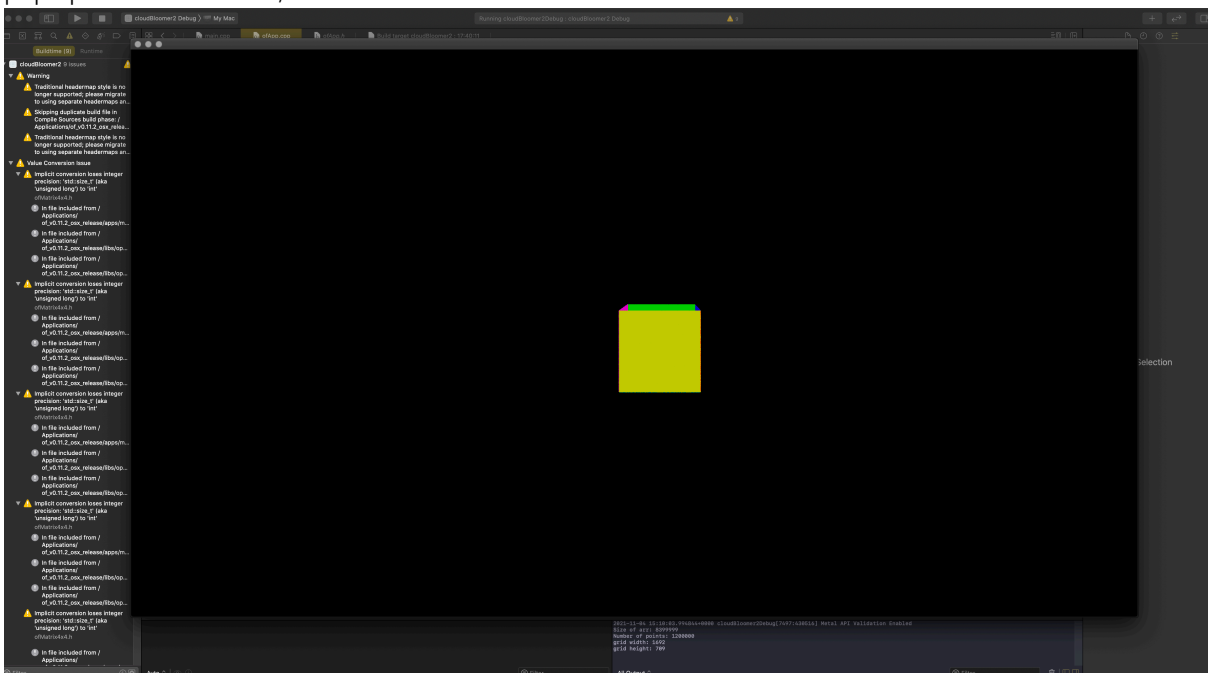
3.3 Build & run

First, make sure that cloudBloomer is in 'live' mode by making sure line 286 is commented out:

```
//ofSaveImage(pix, result);
```

Then click on the RUN button (arrow at the top left of the Xcode window).

You will see Xcode throw up some errors in the sidebar (which you can safely ignore), and then a window will pop up with a static cube, like this:



Note that here the top face of the cube looks transparent (ignore this).

To stop the rendering, press the STOP button (square, next to RUN).

When you're ready to save an image sequence to disk, you simply uncomment line 286 before running:

```
ofSaveImage(pix, result);
```

3.4 Default scaling

Amend lines 9–11 one at a time, changing the scaling factor or offsets by small amounts (5 or 10) and RUN again. These are parameters that you will have to change in order to be able to render point clouds of different sizes to fit the window.

```
float scalingFactor = 70; // try 60 for abstract blender object, 70 for cube
float zoffset = 0;        // how far is the mesh, try 0
float yoffset = 20;       // y offset of the mesh, try 20
float xoffset = 50;       // x offset for the mesh, try 50
```

3.5 Rotations

Line 8 sets the underlying rotation speed.

```
float rotationSpeed = 100.0; // the bigger the number, the slower the rotation
```

Leave it at 100 for now. Let's make the cube spin in different ways. See lines 246–254:

```
  //VARYING ROTATION AROUND DIFFERENT AXES uncomment one or more of lines 247-249
  //ofRotateYRad(ofGetFrameNum() / ((1.1 + (1.3 * a_phase)) * rotationSpeed));
  //ofRotateXRad(ofGetFrameNum() / ((2.1 + (11 * b_phase)) * rotationSpeed));
  //ofRotateZRad(ofGetFrameNum() / ((2.5 + (0.8 * c_phase)) * rotationSpeed));
  //CONSTANT ROTATION AROUND DIFFERENT AXES try in combination with ZOOM line 270
  //ofRotateYRad(ofGetFrameNum() / (1.0 * rotationSpeed));
  //ofRotateXRad(ofGetFrameNum() / (1.3 * rotationSpeed));
  //ofRotateZRad(ofGetFrameNum() / (10 * rotationSpeed));
```

Try uncommenting one of lines 252, 253 or 254 and run; then try uncommenting two or all three lines.

To change the overall rotation speed, amend line 8. To change the relative rotation speeds around different axes, change the multiplier of the variable rotationSpeed in the relevant line. For example, to have the cube rotate twice as fast around the X-axis and three times as fast around the Z-axis as theY-axis, change lines 252-254 as follows:

```
  //ofRotateYRad(ofGetFrameNum() / (1.0 * rotationSpeed));    --- this is our baseline speed
  //ofRotateXRad(ofGetFrameNum() / (0.5 * rotationSpeed));    --- this is DOUBLE the baseline speed
  //ofRotateZRad(ofGetFrameNum() / (0.33 * rotationSpeed));   --- this is THREE times baseline
```

Remember to uncomment the lines that you want to be executed! Also, instead of writing:

```
  //ofRotateXRad(ofGetFrameNum() / (0.5 * rotationSpeed));
```

you could of course write the following, which is mathematically equivalent:

```
  //ofRotateXRad(ofGetFrameNum() * 2.0 / rotationSpeed);
```

While the code is running, pressing the key '6' reverses the direction of all rotations by changing the sign of the variable rotationSpeed (+ to – or vice versa); key '5' increases rotationSpeed by 10% and '4' reduces it. Because of the way the image is rendered, these key presses will cause a 'jump' in the rendering (or in the saved image sequence).

Advanced:

You can also have the rotation speed around any axis vary periodically, by uncommenting lines 247, 248 or 249 (and commenting out the corresponding line 252, 253 or 254). The variables a_phase, b_phase and c_phase are defined at lines 236–238 by periodic trigonometric functions:

```
  float a_phase= 0.73 + 0.49*abs(cos(imageNumber / 160.));//*cos(3.14159*imageNumber/2117.);//tan function for the *_phase variables will send them to infinity
  float b_phase = 0.77 + 0.51 * abs(sin(imageNumber / 270.));
  float c_phase = 0.76 + 0.53 * abs(sin( imageNumber / 101.));
```

The frequency of the trig functions (how often they oscillate) is determined by the frame rate via the imageNumber variable; you can adjust the frequency by changing the number dividing imageNumber in the lines 236–238:

abs(cos(imageNumber/1600.))

will oscillate at one tenth the frequency of:

abs(cos(imageNumber/160.))

Try changing cos or sin to tan and try removing the abs() (absolute value treats negative numbers as positive).

The cos (cosine) function outputs values between -1 and 1, so while the expression cos(imageNumber/160.) returns a floating point number between -1 and 1, abs(cos(imageNumber/160.)) will return a number between 0 and 1.

Similarly, tan(imageNumber/160.)) returns a value between $-\infty$ and $+\infty$, while abs(tan(imageNumber/160.))) returns a value between 0 and $+\infty$.

You can further adjust the rotation by tweaking the numbers added to and multiplying these expressions (in the example below, the numbers 0.76 and 0.53)

```
  float c_phase = 0.76 + 0.53 * abs(sin( imageNumber / 101.));
```

3.6 Zoom/Jitter/Breathe

Lines 256–270 control how the pointcloud is scaled in a periodic manner. Uncomment one of the Breathe lines (257 or 258) for a gentle pulsation; uncomment line 261 for something more jittery, and uncomment one of lines 267–270 for different zooming effects. Suggestions for more advanced manipulation (hyperbolic trig functions, changing zoom speed/scale) are given in the comment lines 265 and 266.

```
  //BREATHING ABS value prevents eversion (flip inside out)
  //ofScale(abs(sqrt(rN * rN2) * b_phase), abs(sqrt(rN1 * rN) * c_phase), abs(sqrt(rN1 * rN2) * a_phase));
  //ofScale(abs(b_phase), abs(c_phase), abs(a_phase));

  //JITTER
  //ofScale(0.5 * rN + rN1, 0.48 * rN1 + rN2, 0.51 * rN2 + rN);

  //ZOOM uncomment one (or more) of lines 267-270 below
  float s_phase = tan(imageNumber / 150.);
  // use line 268 (abs value version) to avoid eversion (flipping back to front); try replacing tan function by sin;
also try sinh, cosh, tanh for different zooms
  // replacing 'imageNumber / 150.' by 'imageNumber / 1500.' in line 264 will make the zoom 10 times
slower; replacing 's_phase' by '10 * s_phase' in lines 267-270 will make the scaling 10 times larger
  //ofScale(10 * s_phase, 10 * s_phase, 10 * s_phase);
  //ofScale(abs(s_phase), abs(s_phase), abs(s_phase));
  //ofScale(10 * s_phase, 10 * s_phase, 10 / s_phase);
  //ofScale(7 * s_phase, 1 / sqrt(abs(s_phase)), 1 / s_phase);
```

Advanced:

You can play with the definition of the variable s_phase at line 264, and of the variables rN, rN1 and rN2 at lines 239–241:

```
  float rN = 1 + 0.005 * (50 - (rand()%100) + 1);
  float rN1 = 1 + 0.0049 * (50 - (rand()%100) + 1);
  float rN2 = 1 + 0.0051 * (50 - (rand()%100) + 1);
```

These last three variables are 'pseudorandom' numbers – as far as we're concerned, they're random numbers generated by the rand() function, and will change with each frame rendered. The expression
(50-(rand()%100)+1)
generates a random integer between -50 and 49.

3.7 Explosion

The explosion effect rearranges the points of the cloud, sending them floating off to a plane. Any rotation or zoom applies to the original point cloud and the plane. Pressing 'C' while the code is running triggers the explosion (and will noticeably slow down the rendering speed, but this won't affect any saved output). Pressing 'V' pauses the explosion. Line 7 sets the default explosion speed:

```
float speed = 1;        // Particle explosion speed
```

Pressing '2' while the pointcloud is exploding accelerates the explosion and '1' decelerates it.

3.8 RGB wraparound distortion/echo

You can create (sometimes interesting) distortions by amending line 96. This changes the way the code parses the pointcloud.

```
    for(int j = 0; j < arr.size(); j+=7) { //change to 7 if alpha or 6 if RGB
```

As above, the code expects each line of the pointcloud data, i.e. each point, to have R, G, B and A (alpha = transparency) values. If you change the number in the expression above to 6, then the code misreads the data and creates R, G and B 'shadows' of the pointcloud object along the axes (this is because it mixes up the colour and coordinate values and reads the alpha value, which is set at 255 = full opacity, as an R, G or B colour value).

3.8 Saving image sequences

To save off an image sequence, change the cloudBloomer code to 'save' mode by uncommenting line 286 to read:

```
ofSaveImage(pix, result);
```

Now when you run the code, it will save a numbered sequence of .png files inside the
/MyApps/cloudBloomer2 /bin/data
folder.

Note that saving frames takes resources, so the window on screen will refresh more slowly, but the saved images will play back at the required frame rate.

Remember to comment out line 286 again when you want to play around with the parameters without saving.

IMPORTANT!
You must move or rename these images before running cloudBloomer again, otherwise they will be overwritten. Alternatively, change the name of the output files at line 282:

```
string line = "output_";
```

Line 280 (which is only a comment line) should read:

```
//pngs will be saved to ../cloudBloomer2/bin/data
```