Video from Image Sequence

You can create a video of your app by saving an image of each frame and then assembling a video from those images afterwards.

- This is a great technique for showing animations and for creating gifs.
- Since you are saving each draw frame, any performance issues due to drawing to the screen will not be noticeable since you will be able to set the framerate of the video during assembly.
- Since there is a lag due to saving frames to disk, this is not so great for showing real-time interaction with your app.

Quick and Dirty

using example 13_00_sequence1

For shorter videos, adding this code to the end of your `draw()` should be sufficient:

```
int frameCount = ofGetFrameNum();

std::string templateFilename = "frame_{FRAME_NUMBER}.png";
auto filename = templateFilename;
ofStringReplace(filename, "{FRAME_NUMBER}", ofToString(frameCount, 4, 0));
ofSaveViewport(filename);
```

This code:

- creates a variable that stores the frame number.
- creates a string that will be our filename template.
- the `{FRAME_NUMBER}` part of our template string gets replaced by our `frameCount` number, with some padded 0's, so that `{FRAME_NUMBER}` always gets replaced by 4 digits (0001, 0002, etc). This becomes important when assembling the frames later.
- We then save the viewport (the app window) to disk with the created `filename` string.

The saved images will show up in the `bin/data` folder.

You will notice that frames will start to be saved immediately and will continue to be saved until you close the app. We will discuss techniques to control starting and stopping the of saving frames in order to capture what you want, but first- assembling the frames.

Assembling frames

The best way to assemble a video from an image sequence that I have found is to use FFmpeg. FFmpeg is a powerful command-line tool for manipulating media, but is initially pretty daunting to work with. To install FFmpeg, see below.

FFmpeg is a command-line utility, so open up your Terminal and `cd` to your `bin/data folder`.

Once there, enter this command to convert an image sequence to .mov

```
ffmpeg -i 'frame_%04d.png' -c:v libx264 -vf "fps=30,format=yuv420p" OUTPUT.mp4
```

This command takes as input each frame in the folder and converts it into an .mp4 video at 30 fps.

Breaking it down:

- `-i 'frame_%04d.png'`: Our input. Frames are added sequentially by filename number and must follow this filename template, where %04d will be substituted our by a 0-padded 4-digit number (remember our ofStringReplace() method!).
- `-c:v libx264` specifies which video codec to use. Here we are using h-264, which compresses at a resonably good quality suitable for the web.
- `-vf "fps=30,format=yuv420p"` creates a 'filter chain'. Here we want the video to be played at 30 fps, and we encode using the color format yuv420p, which makes it compatible for quicktime.
- `OUTPUT.mp4` the output filename. make sure to change this to something memorable, like `funkytime.mp4` or whatever

For better quality (but a larger file size) use:

```
ffmpeg -i 'frame_%04d.png' -c:v libx264 -preset slow -crf 10 -vf "fps=30,format=yuv
```

- this encodes at a slower rate (less mistakes!), and allows for a higher bit-rate (more data per frame!).

Seamless Gifs

using example 13_00_loopyGif01

The technique for creating seamless gifs comes from necessarydisorder

We will build off of the quick and dirty method. We will set a max amount of frames to capture. If our frame count is under this max we will record frames. If our frame count is over this max we will stop recording frames. All our animations will be based on the ratio of our current frame and the max frame (how far along we are to hitting that max frame).

in our draw() add:

```
int numFrames = 180;
int frameCount = ofGetFrameNum(); // same as before
float t = 1.0 * frameCount / numFrames; // t goes from 0.0 - 1.0!
```

`float t` is important, as it will be a function in all the animations. I'll talk about it in a second, but first wrap the the code that saves the frame in an `if` statement. So that is looks like:

```
std::string templateFilename = "frame_{FRAME_NUMBER}.png";

if (frameCount <= numFrames)
{
    auto filename = templateFilename;
    ofStringReplace(filename, "{FRAME_NUMBER}", ofToString(frameCount, 4, 0));
    ofSaveViewport(filename);
}
else
{
    ofLog() << "Saving Frames Completed!";
}
```

Let's play around with this with a simple animation. We will draw a circle rotating around center.

In `draw()`, add code to draw a circle:

```
float radius = 100.0;
float xPos = radius * cos(TWO_PI * t);
float yPos = radius * sin(TWO_PI * t);

ofPushMatrix();
ofTranslate(ofGetWidth()/2, ofGetHeight()/2);
ofFill();
ofDrawCircle(xPos, yPos, 5);
ofPopMatrix();
```

The `cos()` and `sin()` functions control the movement of the circle. They output a number

between -1 and 1, which we scale by `radius`, so `xPos` will be between -200 - 200. `cos()` and `sin()` repeat every `TWO_PI` (cos(0) = 1 = cos(TWO_PI)) and since our input to `cos()` and `sin()` is scaled by `t`, we will reach `TWO_PI` only when we hit `numFrames`.

For clarity, our entire `draw()` should look like:

```
void ofApp::draw() {

    ofBackground(0);

    int numFrames = 180;
    int frameCount = ofGetFrameNum();
    float t = 1.0 * frameCount / numFrames;

    float radius = 100.0;
    float xPos = radius * cos(TWO_PI * t);
    float yPos = radius * sin(TWO_PI * t);

    ofPushMatrix;
    ofTranslate(ofGetWidth()/2, ofGetHeight()/2);
    ofFill();
    ofDrawCircle(xPos, yPos, 5);
    ofPopMatrix();

    std::string templateFilename = "frame_{FRAME_NUMBER}.png";
    if (frameCount <= numFrames)
    {
        auto filename = templateFilename;
        ofStringReplace(filename, "{FRAME_NUMBER}", ofToString(frameCount, 4, 0));
        ofSaveViewport(filename);
    }
    else
    {
        ofLog() << "Saving Frames Completed!";
    }

}
```

It's important to note that the circle makes one revolution every `numFrames`. If we would like to speed up or slow down the circle than we can change the value of `numframes`, or add a scaler to the cos/sin functions, such as `cos(TWO_PI * t * speed)`. If `float speed = 2` then our circle would make two revolutions every `numFrames`.

Once you have those frames, make them into a movie using FFmpeg.

```
ffmpeg -i 'frame_%04d.png' -c:v libx264 -preset slow -crf 10 -vf "fps=30,format=yuv
```

Then, once you have your video, you can convert it into a .gif with:

```
ffmpeg -i INPUT.mp4 -filter_complex "[0:v] split [a][b];[a] palettegen=stats_mode=s
```

Be sure to use the correct input and output file names.

You can read all about this command here but basically in order to encode super great looking gifs you need to sample every frame and create a color palette, and then apply that palette to every frame of the gif. This is more important for colorful gifs.

You can also change the fps and scale when encoding to gif:

```
ffmpeg -i INPUT.mp4 -filter_complex "[0:v] fps=12, scale=w=480:h=-1,split [a][b];[a
```

note When encoding to .mp4 I usually set the fps to 30, so everything moves half the speed. This is, of course, optional.

Start Recording on key press

If you would like to trigger recording with a particular event (such as a key press), you can create a global `boolean` that manages when you should record, such as if `boolean bIsRecording = false` no frames are captured, and `bIsRecording = true` recording happens. You can still convert to gifs with this method, but I wouldn't recommend it for making seamless gifs.

Take a look at 13_01_ImageSequence_keyPress for how to capture the save to camera on key press.

Screen Capture

Instead of saving frames from our app, we will screen capture the entire screen.
There are a numerous applications that will screen grab your screen, including Quicktime Player, which can capture a particular area, and FFmpeg. Using Quicktime Player and converting the video afterwards is surely a way to go, but I will walk though the FFmpeg way.

FFmpeg can only record the entire screen on macOS. Here is the code:

```
ffmpeg -f avfoundation -i "1:none" -c:v libx264 -crf 0 -preset ultrafast -r 30 -vf
```

- in this example, my app window is 640x640.
- Run your app and then stick the app window in to top left corder of the screen.
- Then run the FFmpeg script to start recording the entire screen.
- hit 'q' to stop recording, close your app
- crop and scale the output video, re-encode fro smaller filesize.

```
ffmpeg -i output.mp4 -c:v libx264 -crf 16 -preset veryslow -vf "scale=w=iw/2:h=ih/2
```

Reading: https://trac.ffmpeg.org/wiki/Capture/Desktop

OR just use Quicktime player to screen grab and then re-encode the video with FFmpeg using:

```
ffmpeg -i INPUT.mov -crf 10 -vf “scale=640:640, setpts=1.0*PTS” -c:a copy -tune gra
```

Resources

To install FFmpeg:

- macOS
  - Best way is through homebrew, a mac package manager.
  - First, install homebrew. Copy and paste into Terminal: `bash /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"`
    - Check out their installation page
    - After homebrew has installed, run `brew update` twice.
  - install ffmpeg: `bash brew install ffmpeg`
  - You're done. Run `ffmpeg -version`

    - Take a look at FFmpeg's installation guide for more options.

- Windows

    - Someone distributes a build at https://ffmpeg.zeranoe.com/builds/. Awesome.
    - If that doesn't work, good luck! https://trac.ffmpeg.org/wiki/CompilationGuide/MinGW

- Video from image sequence

- .gif from video

- Encoding tips

- Daily Sketches