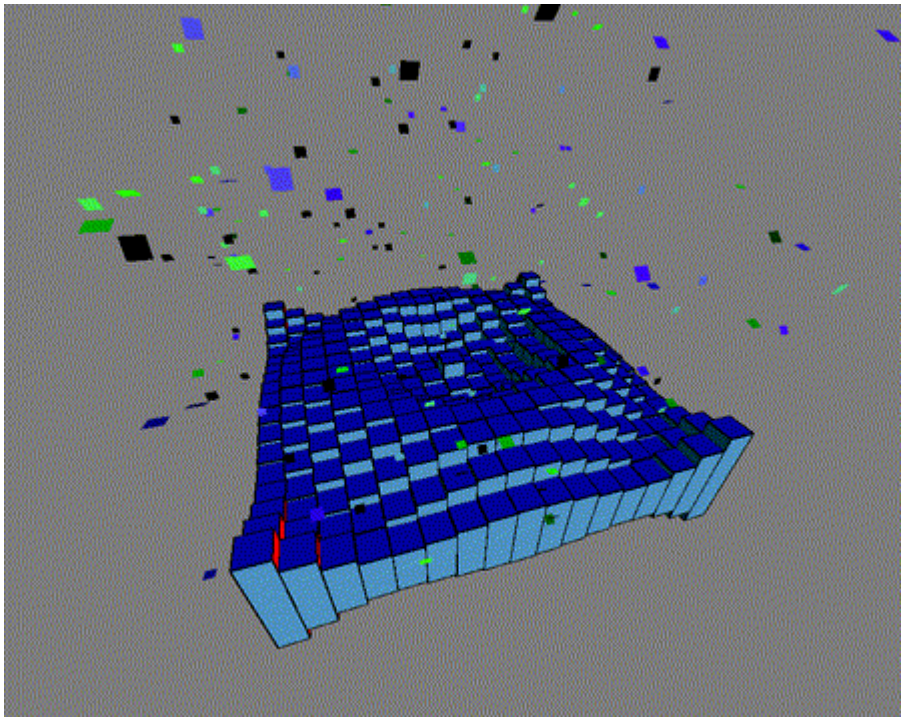
3D sine games

coursera.org/learn/uol-graphics-programming/supplement/pUC9c/3d-sine-games



3DSineGames-Template

ZIP File

In this assignment the goal is to replicate this pulsating 3D structure as shown in the image above.

Please note: This is a graded assignment. This is a draft of the assignment made available before course work submission is open, only minor changes may occur.

Where do I submit?

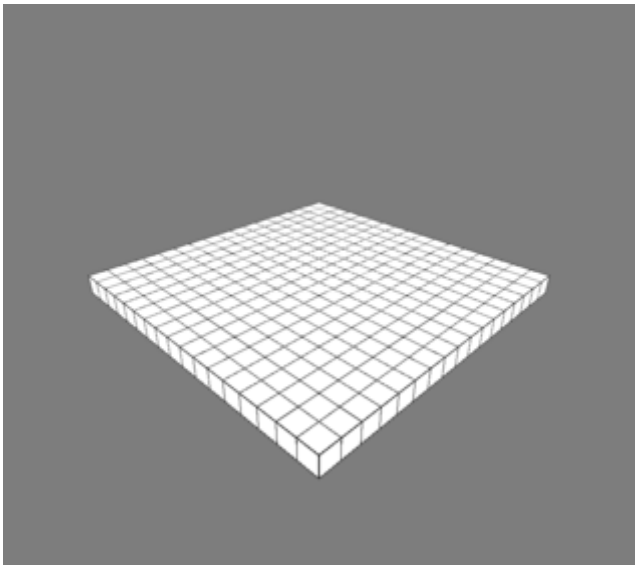
There are eight graded assignments like this on the course. This one and three more make up the final assignment and are submitted in week 20. Nevertheless, we strongly recommend that you complete this graded assignment in the week it has been assigned and not wait for the end of term submission. You need to master the material included in these before you continue with the next weeks. Once you complete it, save it somewhere safe, do not share online using the webspace and upload it with the others when prompted in week 20.

Steps to complete

Start by downloading the **3DSinGames-Template** folder.

Step 1: Using a nested for loop, create a grid of boxes of size 50x50x50 from -400 to 400 in the x-axis and -400 to 400 on the z-axis. Place the camera at location (800, -600, 800) and have it point at the centre of the scene. If you've done things right you should be seeing something like the image below.

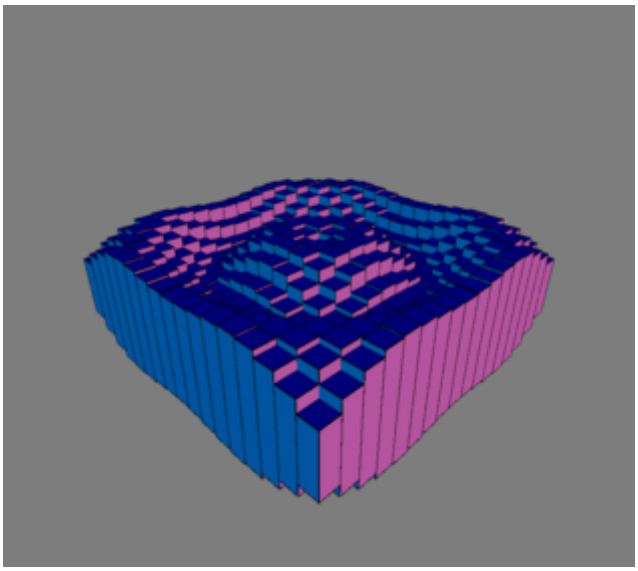
Step 2: Set the material to normal, set the stroke to zero and use a stroke weight of two to better distinguish the boxes.



Step 3: For each box in the nested for loop, calculate its distance from the centre of the coordinate system using its x and z coordinates and dist(), then save it in a variable called *distance*.

Create a new variable *length* and modulate its value from 100 to 300 using the sin() function and the distance variable. Use the *length* variable to set the height of the boxes. If you have done things right you should have a wavy structure like the one below with boxes ranging in height from 100 to 300 depending on their distance from the centre. Add *frameCount* to *distance* in order to animate the wave.

Step 4: Amend the camera() command and get the camera to fly in a circle around the structure we have created, as it does in the animation at the top of the page. The animation at the top resets after 10 seconds. Your camera should fly continuously, uninterrupted around the object. (Hint: look at the video lecture on moving the camera for how to do this).



Step 5: Time for confetti! Create two global arrays, one called *confLocs* to store the location of each confetti and one called *confTheta* to store the initial angle of each confetti.

In the setup() function use a for loop to push 200 3D vectors into *confLocs*. Make the x component of the vector have random values ranging from -500 to 500, the y component from -800 to 0 and the z component from -500 to 500. This way we'll have spread confetti all over the structure.

Push also a random angle from 0 to 360 onto the *confTheta* array. Create a function called *confetti()* where you'll loop over the *confLocs* array. For each entry translate to that location the 3D vector describes, rotate by the corresponding theta and draw a plane of size 15x15. Remember to apply these transformations within a push()/pop() pair so that it looks right.

Step 6: Let's animate the confetti! Increment the y-coordinate of the specific confetti by 1 so that it keeps travelling downwards, and increment the rotation by 10 so that it keeps spinning. At the bottom of the for loop add an if statement to check if the y-coordinate of the confetti is greater than 0, that is, if it has reached the middle of the coordinate system. If it has, set the specific vector's y component to -800, so that the confetti starts at the top of our world. Leave the other two components intact.

Step 7: Make the sketch your own by implementing two of the ideas for further development. If you choose to implement different materials please leave the code for step 2 (material and stroke weight) commented out in your code submission. Keep performance and frame rate in mind for all code implementations. Points given to ambitious learners.

Ideas for further development:

- Add 2D noise to your sin wave. HINT: this will involve amending your code in step 3, look at the noisy grid assignment for clues about how to do this.
- Customize the sketch by adding different materials that are affected by lights and add lights. Look at the p5.js documentation on lights and materials if needed. Can you add a different material only to the cubes and not the confetti? HINT: think back to push() and pop().
- Create some p5.js sliders to make some of your variables for the cube grid dynamic, for example the height of the cubes or the speed of the sine wave or potentially resolution of the 2D noise.

Marking Rubric

Step 1 - [2 points]: A grid of tiles of the right size, spread over the right area, has been produced.

Step 2 - [1 point]: Correct material and stroke is on display.

Step 3 - [2 points]: Structure is wavy like the demo at the top of the page.

Step 4 - [1 point]: Camera flies around like in the demo at the top of the page.

Step 5 - [2 points]: Confetti appear on top of the structure like in the demo, at random locations and random angles, but do not necessarily animate.

Step 6 - [1 point]: Confetti is falling downwards and is also rotating. When it reaches 0 on the y axis it resets to the top.

Step 7 - [3 points]: Has the student implemented ideas for further development?

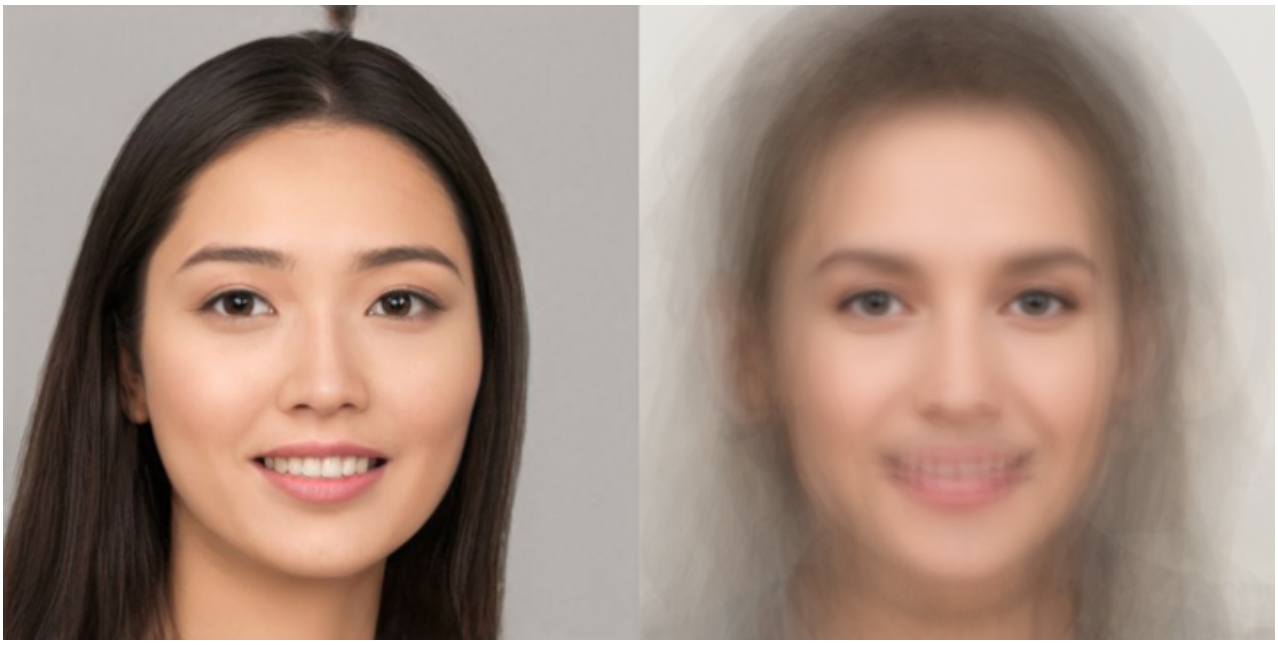
Average face

coursera.org/learn/uol-graphics-programming/supplement/QDDOb/average-face

Some computational artists build whole careers based on averaging images. Jason Salavon created a whole series of special moments, which can be found here. Below are *Newlyweds* and *Kids with Santa,* two examples from the series, used with artist's permission.

In this assignment we are going to use a data set of 30 faces. These faces are part of 100,000 faces belonging to people that have never existed. They have been generated using machine learning by a team of AI and photography professionals. More information on their work can be found here. The images used in this assignment have been used with permission by the authors.

Your goal in this assignment is to calculate the average face from these 30 images as seen in the image on the right below.



Start by downloading the **averageFace-Template** which contains the required data set.

averageFace-Template

Warning: Remember to use Bracket's live view to view your sketch. Loading images will not work when the html is viewed independently in the browser.

Please note: This is a graded assignment. This is a draft of the assignment made available before course work submission is open, only minor changes may occur.

Where do I submit?

There are eight graded assignments like this on the course. This one and three more make up the final assignment and are submitted in week 20. Nevertheless, we strongly recommend that you complete this graded assignment in the week it has been assigned and not wait for the end of term submission. You need to master the material included in these before you continue with the next weeks. Once you complete it, save it somewhere safe, do not share online using the webspace and upload it with the others when prompted in week 20.

Steps to complete

Step 1: Let's load the faces in memory. Inspect the assets folder. Notice there are 30 images with names starting from 0.jpg to 29.jpg. Use a for loop within the preload() function to load all 30 images into the *imgs* array. (Hint: Create a string called *filename* made up from the path to the images, the for-loop index and the file extension. You can use console.log to make sure you've built the right *filename* string.)

Step 2: Update the createCanvas() line to create a canvas twice the width of the first image in the array, and equal to the first image's height. Draw the first image on the left of the canvas. If you've done things right you should have one of the faces on the left and a grey area of equal size on the right.

Step 3: In the setup() function initialise the *avgImg* variable using the createGraphics() command. Set its size equal to the size of the first image in the array. This way, we have created an empty buffer to save the results of our calculations.

Step 4: Next, we need to access the pixel data of all the images in the *imgs* array and the variable *avgImg*. In draw(), use a for loop to call the *loadPixels()* command on all images within *imgs*. Also call *loadPixels()* on the *avgImg* variable.

Step 5: Create a nested for-loop looping over all pixels on the first image in the array. Convert the x and y coordinates from the for-loop to a pixel index value and use that value to set the corresponding pixel in the *avgImg* to red.

After exiting the nested for loop, update the pixels of the *avgImg* to let p5js know that the image has had its data changed, and draw the *avgImg* to the right of the existing image. If you've done things right, the left side of the canvas should have the face of the first image in the array and the right side should be bright red.

Also add a noLoop() at the end of the draw() function as the calculations we are about to do are intense and we only really need to do them once. No need for looping.

Step 6: Inside the nested for loop, create three variables *sumR, sumG, sumB* and initialise them to 0. This is where we are going to store the sum of each channel for that pixel. Create a for-loop just under these variables, looping through all the images in the *imgs* array and for each channel add its value to the corresponding sum variable. Just under this for-loop update each channel in the *avgImg*. (Hint: You'll need to use the sum variables as well as the size of the *imgs* array.) If you've done things right you should be seeing the average image as displayed at the top of the exercise.

Step 7: Can you extend the sketch by implementing the two ideas for further development? HINT: Do not remove the noLoop() from the end of draw(), simply call loop() at the end of the user input functions mentioned below.

Ideas for further development:

- How would you change the code so that the image drawn on the left is a random face from the array of faces rather than just the first one, with a new random face selected using the keyPressed() function?
- On mouse moved could you have the pixel values of the second image transition between the randomly selected image and the average image based on the mouseX value? HINT: Use the p5 lerp() function, read the documentation to understand what you need to do.

You only need to submit the final version of your code after completing the steps. The step by step output in the rubric should only serve as a guide to understand what output should have been achieved at each step.

Marking rubric

Step 1 - [2 points]: Images loaded successfully using a for-loop (check code).

Step 2 - [1 point]: Face appears on the left, grey canvas on the right.

Step 3 - [1 point]: Image initialised correctly within setup() function (check code).

Step 4 - [1 point]: Images are looped over and loadPixels() is called on them.

Step 5 - [2 points]: Face appears on the left, and the right side of the canvas is red. Conversion from 2D to 1D coordinates has taken place (check code).

Step 6 - [2 points]: Average image appears on right side of the canvas.

Step 7 - [3 point]: Points awarded based on whether the solutions to the ideas for further development where correctly implemented.

Your own Instagram filter

coursera.org/learn/uol-graphics-programming/supplement/zGGMg/your-own-instagram-filter

instagramFilter-Template

ZIP File

Instagram has a large collection of filters. In this assignment we'll try to create our own, copying features from a few of them in order to achieve the effect you see in the image above.

Please note: This is a graded assignment. This is a draft of the assignment made available before course work submission is open, only minor changes may occur.

Where do I submit?

There are eight graded assignments like this on the course. This one and three more make up the final assignment and are submitted in week 20. Nevertheless, we strongly recommend that you complete this graded assignment in the week it has been assigned and not wait for the end of term submission. You need to master the material included in these before you continue with the next weeks. Once you complete it, save it somewhere safe, do not share online using the webspace and upload it with the others when prompted in week 20.

Steps to complete

Start by downloading the **instagramFilter-Template** folder.

Warning: Remember to use Bracket's live view to view your sketch. Loading images will not work when the html is viewed independently in the browser.

The template code provides the general outline for how this filter will work. We'll call this filter the early bird filter because it looks a bit like the one by Instagram. Notice how the filter itself is a function called earlyBirdFilter() which takes an image as an input and returns an image as output. When you run the code the first time you should see the original picture of the boy with the Husky on the left and the rest of the grey canvas on the right. As you work through the code you'll be uncommenting each line in the earlyBirdFilter() function and implementing the sub-filters that make it up. Filters in social media apps are often combinations of simpler filters we have already examined.

Important: Please use the image we provided in order for us to be able to compare results and be able to mark you. Let's get to it!

Step 1: Inside earlyBirdFilter() function, uncomment the line that calls the sepiaFilter(). Implement the sepiaFilter() function. The basis of this filter is one of the simple filters we saw in class (e.g. invertFilter). Copy the code over and modify it to turn the image into sepia. A simple implementation involves the following conversion of each channel of all pixels.

```
newRed = (oldRed * .393) + (oldGreen *.769) + (oldBlue * .189)

newGreen = (oldRed * .349) + (oldGreen *.686) + (oldBlue * .168)

newBlue = (oldRed * .272) + (oldGreen *.534) + (oldBlue * .131)
```

Importantly, make sure to constrain *newRed, newGreen* and *newBlue* to values between 0 and 255. Remember to make sepiaFilter() return the resulting image. If you've done things right you should be seeing the image below.

Step 2: Adding dark corners, also known as vignetting, will give our image a slightly older feel. Uncomment the next line in the earlyBirdFilter() that calls the darkCorners() function and implement the corresponding filter. The basis of this filter is a simple filter like the one above. What you'll need to do is adjust the luminosity/brightness of the pixel by scaling each colour channel. Pixels that are:

- up to 300 pixels away from the centre of the image – no adjustment (multiply each channel by 1)
- from 300 to 450 scale by 1 to 0.4 depending on distance
- 450 and above scale by a value between 0.4 and 0

Hint: You'll need to use the map() and constrain() functions in order to remap the distance of each pixel to a new variable called *dynLum* (for dynamic luminance) which will hold the scaling that will be required for each channel. If you've done things right you should see an image like the one below.

Step 3: Uncomment the next line in the earlyBirdFilter() that calls the radialBlurFilter() function and implement the corresponding filter. The basis of this filter is the blur filter we saw in class. However, this time we'll create a radial filter that blurs more as you move away from its centre. We'll also use a bigger kernel, as you have seen at the top of the template. Copy over the blur filter and the convolution function from the examples demonstrated in class. The main difference will be in how the new values are calculated. Just after the convolution call inside the blur function, we would need to update the each channel of the specific pixel as we're doing below for the red channel:

```
imgOut.pixels[index + 0] = c[0]*dynBlur + r*(1-dynBlur);
```

where *c[0]* is the red channel returned from the convolution, *r* is the red channel in the original image and *dynBlur* is a value we generated using the distance from the mouse. For each pixel we need to calculate the distance between it and the mouse **on the colour image**. We need to remap the distance from a range 100 to 300 to a new range from 0 to 1. We then need to constrain the returned value from 0 to 1 and save it in the *dynBlur* variable.

What the *dynBlur* variable allows us to do when used in the operation above is to say how much of the blur we want to use. When the pixels are up to 100 pixels from the mouse they are clear (the original image values are used and none of the ones returned from the convolution). As the distance from the mouse increases from 100 to 300, more of the blurred image is gradually used until 300 is reached - after which only the blurred image is used and none of the original (i.e. clear) image.

Please note: because this operation is slow we have used the noLoop() command inside the draw() function therefore the mouseX and mouseY are not updated until you click somewhere. Click on the part of the colour image you want to place at the centre of the radialBlurFilter().

If you have done things right you should see something like the image below. I clicked on the face of the boy in the colour image. Notice the radial blur centered around there.

Step 4: The final touch: Let's add a border around the image. To do this we'll have to implement the borderFilter() function. Like all of the above filters this one will also take an image *img* as an input and then create a local buffer called *buffer* of the same size as the input image (Hint: you'll do this using the createGraphics() command). Draw the *img* onto the *buffer*. And draw a big, fat, white rectangle with rounded corners around the image. See the documentation about drawing rectangles with round corners here. This should create a rectangle like the one seen below. Make sure you return the *buffer* at the end of the function.

Draw another rectangle now, without rounded corners, in order to get rid of the little triangles so you end up with the image below.

Step 5: Can you further develop the sketch by implementing some logic to switch between different filter effects on the second image? On key press change between a set number of different “filters”, perhaps try to combine different kernels (stored in the variable called “matrix”) with different pixel color effects like the sepia color effect, for example grey scale is another effect. Make sure that the first effect visible when the sketch is loaded is the sepia effect with the radial blur and dark corners as per the above instructions, do not change the logic of applying the radial effect on mouse pressed. Beneath the images include written instructions about which key or keys to press. Include comments in the code about what you have done. Points awarded to ambitious learners.

Marking rubric

Step 1 – [2 points]: Sepia filter has been implemented successfully and images look very similar to the ones provided by the instructor.

Step 2 – [2 points]: Vignetting has been achieved using the map() and constrain() functions (code check required) and results look very similar to the ones provided by the instructor.

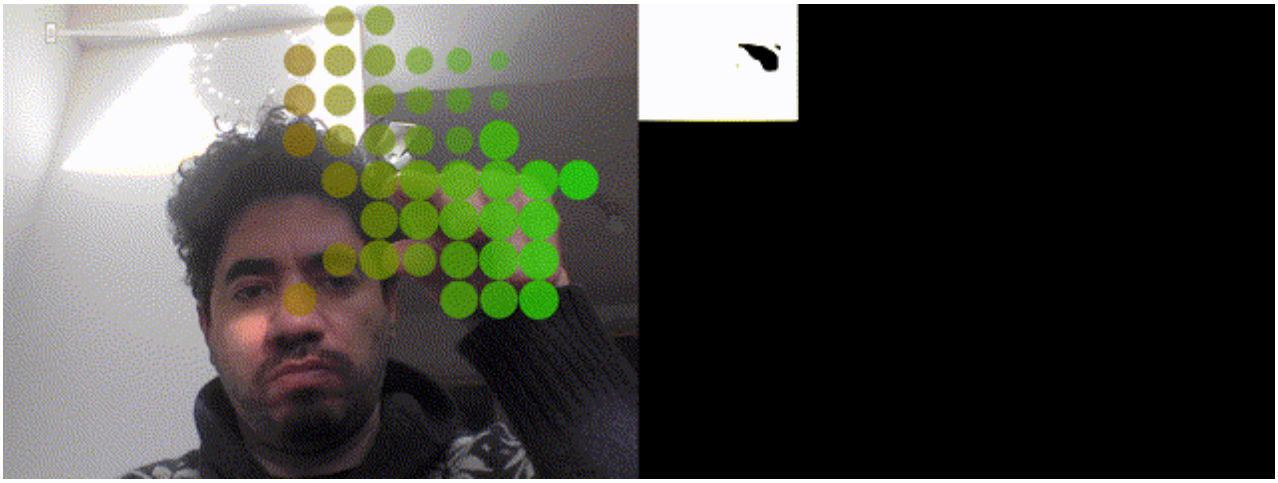
Step 3 – [3 points]: Radial blur has been achieved using the map() and constrain() functions (code check required). Clicking on the face of the boy in the colour image replicates the results provided by the instructor.

Step 4 – [2 points]: Borders recreated using the technique suggested by the instructor.

Step 5 – [3 points]: Did the student successfully and clearly implement functionality to switch from the main filter effect to some other filter effects?

Webcam piano

coursera.org/learn/uol-graphics-programming/supplement/GoUUj/webcam-piano



Grid

ZIP File

In this assignment the goal is to replicate the computer vision-based interaction seen in the image above. The assignment is a simplified version of the work done by Memo Akten in his *WebCam Piano 2.0*. More information on that can be found here.

Please note: This is a graded assignment. This is a draft of the assignment made available before course work submission is open, only minor changes may occur.

Where do I submit?

There are eight graded assignments like this on the course. This one and three more make up the final assignment and are submitted in week 20. Nevertheless, we strongly recommend that you complete this graded assignment in the week it has been assigned and not wait for the end of term submission. You need to master the material included in these before you continue with the next weeks. Once you complete it, save it somewhere safe, do not share online using the webspace and upload it with the others when prompted in week 20.

Steps to complete

Your starting code should be the **backgroundSubtraction** example code provided in class. We have to turn this example from a background subtraction example to a frame differencing one.

Step 1: Start off by renaming the *backImg* to *prevImg* everywhere in the code so that it's more intuitive for frame differencing. Run the code and make sure it runs just like before.

Step 2: In background subtraction whenever we press a key we save the current snapshot to use as a background. In frame differencing we want save the previous frame in every loop. To do that, move the code updating the *prevImg* from the keyPressed() function to the end of the draw() loop. If you've done things right you should have a working copy of a frame differencing example. The image on the right should show you where the movement in the frame is.

Step 3: Download the Grid.js file file included on this page. This file contains the class that allows for the circles to be drawn on screen where there's movement. Include a reference to it in the index.html file, create a global variable called *grid* and initialise it in the bottom of the setup() function by calling this line:

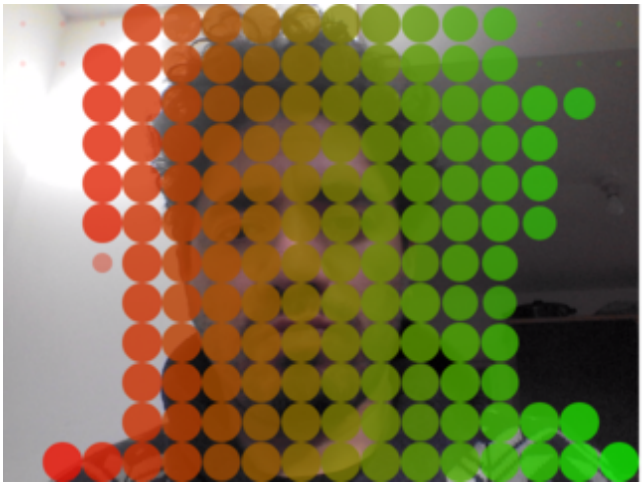
grid = new Grid(640,480);

and at the bottom of the draw() function call this line:

grid.run(diffImg);

If you've done things right the grid should appear on top of the colour image on the right wherever movement is detected, like in the image below. However, notes are activated even in the areas where tiny bits of movement (or even noise) triggers them. This won't make for a very good webcam piano. We'll fix that next.

Step 4: If your camera is anything like mine you should notice that there's a lot of activation of the "notes" of the piano even when we don't quite intend it. Playing with the threshold can't really fix it. Let's address this by running the blur filter on the *currImg* at the top of the draw() function, right after we have copied the contents of the video into it. See the filter documentation here. Use the BLUR filter with 3 iterations (i.e. pass 3 as a 2nd parameter in the filter function). If you've done things right you should see something like the image below where the noise is removed due to the blur, but the sketch will be running very slowly. We'll fix that next.



Step 5: Right before using the blur filter on the *currImg,* use the resize command to scale it down to a quarter of the size it was. Documentation here. Make sure you do the same for the *diffImg* a couple of lines later or things will break! The sketch should now be running quite fast (>40fps) and your sketch should look like the animation at the top of this page.

Step 6: Extend the sketch. This is your last graded assignment and is your chance to make use of techniques and knowledge learnt throughout the course. Select two ideas for further development aiming to write clean and modular code. Include comments about how and why you extended the sketch in the way you did. Get inspiration from Memo Atken's webcam piano here.

Ideas for further development:

- Customize the graphics. In Grid.js you could customize the base grid of graphics thinking about color, opacity or shape. Can you make use of *noteState* to drive different effects that change over time after the note has been activated?
- Trigger secondary graphics effects or animations when an active note is drawn. Can you include rules that only trigger the effects sometimes, perhaps making use of noise or randomness?
- Implement the core p5.js sound library to play sounds depending on which "note" in the grid is activated. You should read and use the p5.js documentation about starting the audio context on user interaction to make sure your audio works across all browsers.
- Implement a custom "Note" class that is used in Grid.js. Instead of an array of values for *noteSize*, *notePos* and *noteState* you would have an array of notes. Think about what parameters you would need for the Note constructor method and what other methods the Note class would need, both the methods needed to adapt the existing functionality from the Grid.js code and any custom methods you would like to add.

Marking rubric

Step 1 - [1 point]: Renaming of *backImg* to *prevImg* has taken place.

Step 2 - [2 points]: Frame differencing implemented by moving *prevImg* around.

Step 3 - [2 points]: Learner has included Grid.js correctly and grid activates with movement.

Step 4 - [2 points]: Learner has included blur in order to reduce the amount of noise that activates the grid.

Step 5 - [2 points]: Learner has scaled down images processed (*currImg*, *diffImg*) so that the sketch runs fast after blurring has slowed it down.

Step 6 - [3 points]: How much has the learner extended the sketch? Learner has included comments about the extension and shows understanding of techniques learnt throughout the course.