Illustrative Computer Graphics: Exercise

Programming Tasks and Conventions

- Practical part of this course:
  - Programming tasks related to lecture material
  - You get the basic framework of a sketch via the Gitlab repository
  - New sketch each week, grading info in the repository
  - **Practical results will be shown in the exam!**
- Reference results included in the giveaway
- Fill in sections marked with „TODO“ in the giveaway code
- Processing
  - Download from www.processing.org
  - Java based, with extensions for graphics
  - Read the docs, we don‘t cover all the details

Git

- Version Control system
- Can handle files an nonempty(!) Folders
- Workflow
  - Git pull: Gets file from the remote repository to your local copy
  - Git add: Adds a new local file
  - Git commit: Captures changes to tracked files in the repository. DOES NOT UPLOAD!!!
  - Git status: Shows the current status
  - Git push: Actually uploads all commits to the default remote
  - Git merge: Combines two commits into one, while checking for conflicts
- Our Repository:
  gitlab.inf.uni-konstanz.de/visual-computing/lecture-illustrative-computer-graphics/icg-exercise-2021

Lecture Prerecordings

- No recordings, unless we need to switch to online
- Uni-Cloud
  - Link https://cloud.uni-konstanz.de/index.php/s/wtwP4RDzrKKmssL
  - Also on Website
  - Password = `Icg_2021!`
  - Also: Lecture slides

Basics of Processing

- Subset of Java (some features missing)
- Primarily designed to easily render basic graphics
  - Prototyping
  - Data visalization
  - Education
  - Art
  - Not suitable for high performance
- Very popular among artists and designers
  - Beginner friendly :)
  - But beware of bad programmers in the forums!
- Books if you are interested: https://processing.org/books/
- You can run processing in eclipse if you want
  - Ok for this course if you handle it yourself and we can see results easily

Basics of Processing: Draw

- Special function: Gets called for every frame
  - `background([color])` to clear the window
  - Calls to functions like `line()` or `ellipse()` draw directly to the window
  - We commonly draw to an image and use `image()` to put it on screen
- Controlling `draw()`:
  - `noLoop()`: Disables `draw()`, no key events recorded etc. You need to call `draw()` manually in your own loop
  - `loop()`: Enables `draw()` again
  - `framerate()`: Controls how often `draw()` gets called per second

Settings and Setup

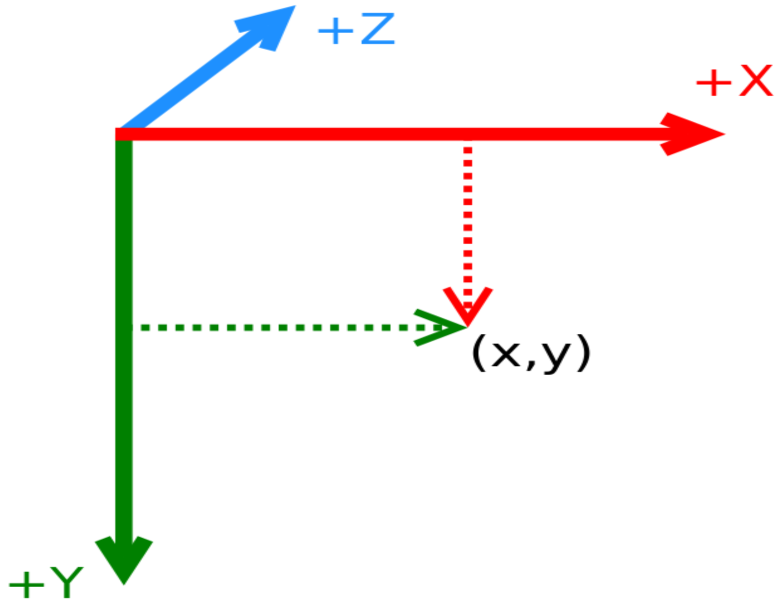
- `Setup()` is run once at sketch startup to intialize everything
  - Inital `background()` for an empty screen
  - Setting up drawing functions like `noStroke()`, `fill()` etc.
  - Initial run of some functions to immediately show a result
  - Do not call `setup()`again, it does extra things in the background (implement a reset() method for yourself if needed)
- `Settings()` is run before `setup()`
  - `size(width, height)` sets the window size
  - Works in setup with fixed window size like `size(200, 200)`
  - Cannot be run in `setup()` with variables like `size(imageWidth, imageHeight)` for… reasons.
  - In settings you cannot use most processing functions (`loadImage()` works despite the documentation claiming it doesn‘t).

Misc

- KeyPressed()
  - Called when a key is pressed, use `if(key == ‘x‘) {…}` to run something
- Mouse input:
  - mouseX and mouseY are global variables and contain the current pixel the mouse is pointing at
  - If mouse is outside of window: 0 or last known location
  - mouseClicked() (and others) are functions like keyPressed(), which you can add your own code to
- When working with Pimage.pixels calls to loadPixels() and updatePixels are required to make changes appear.
- PGraphics is just a wrapper for PImage and can be used interchangeably in some contexts (read docs for when).
- Parameters are passed by value, but these values are references! I.e. you can pass a Pimage and changes made will be visible in the outer scope. Setting the reference to null has no effect though.

Conventions

- RGB is a triple of values [0; 255]
- Intensity is a floating point value in [0.0; 1.0]
- Images are accessed at point (x, y) via img[x][y]
- Float[][] intensityImage access
  - intensityImage[x][y]
  - Stored as array(y) of arrays(x)
  - intensityImage[0] → first column along Y
- Coordinate System: Top left zero, Right hand



Debugging Hints

Debugging

- Removal of defects in your program
- Common failure modes in graphics programming:
  - No output
  - Broken output
  - Small details
- Regular mode: Code & Fix
  - Works on small sketches but frustrating
  - No guarantee of success
- Suggested methodology for his course

Scientific Method

1) Observe
2) Form Hypothesis
3) Make predictions
4) Test prediction by experiment
5) Repeat 3 & 4 until hypothesis matches observation well

Scientific Debugging

1) Observe failure
2) Create a hypothesis for cause of failure (or at least the general region)
3) Use hypothesis to make a prediction
4) Test the hypothesis by performing an experiment that makes the cause of the error observable

Observing failures

- **Read the error message.**
- Make your code easy to trace!
  - Avoid complicated, long arithmetic, or logic statements
  - Build it with debugging in mind
- Make your code easy to understand
- Crash early
  - Detect NULL image after loading rather than during painting.
- Avoid side-effects and shared data (within reason)

Forming a hypothesis

- Reason from first principles and facts
- Avoid assumptions, even if they seem reasonable
  - a+1 < a for all natural a
  - System clock time advances consistently
  - I have opened the correct file
  - Image files always contain at least one pixel
- If you can’t find a testable hypothesis, eliminate parts of the program or bisect it to narrow it down a possible cause
- Formulate a concrete hypothesis and make sure you understand what is going on
  - Data layout, coordinate system etc. No guessing!

Make a Prediction

- Examples:
  - “The last row of pixels in the image is never examined”
  - “There is a missmatch between the range I use (0.0 – 1.0) and what Processing expects (0 – 255)
  - “The loaded image is not oriented as I expect”
- Should be concrete and testable
- If you can’t find one, you don’t understand the problem enough yet → Observe more.

Test the Hypothesis

- Examples:
  - “The last row of pixels in the image is never examined”
    - Have a copy of the image and color every visited pixel red.
  - “There is a missmatch between the range I use (0.0 – 1.0) and what Processing expects (0 – 255)
    - Print statement or debugger view. Take note of any casting you might have done. Maybe iterate lal pixels and check for out-of-range values (Unexpected NaN?)
  - “The loaded image is not oriented as I expect”
    - Load a 3x3 test image and look at the values (maybe the image loader/writer is doing some transformation?)
- The more primitive, the better.
- Be wary of your test altering program behaviour (Heisenbugs)
- Keep testing code around for later

General Rules for Debugging

- Don‘t make random changes
- Don‘t change more than one thing between observations
- Build your code to be easily debuggable
  - Long conditionals or lots of side effects are terrible
- Avoid complexity whenever possible
- Use the debugger
- Use print statements even more (and keep them in your code)
- Use units! Write down the coordinate system you are using
  - `StrokeLength += SegmentLength // Seems legit.`
  - `StrokeLengthMM += SegmentLengthPixels // Obviously wrong!`
- Use actual descriptive variable names
  - `arr[i][j][c] // ?!`
  - `intensityValueArrayNormalized[xPos][zPos][yPos]`
- Put everything under source control

Processing Warumup

- Create ICG/warmup/warmup.pde
- Write setup()
  - Set window size to something suitable for your display
  - Set framerate to 30
- Write draw()
  - Reset background
  - Draw a square moving according to some rule
  - Draw some objects of random color
  - Draw one ellipse orbiting another
  - Place some text
- Add some of your own ideas to familiarize yourself with the language
- https://processing.org/reference/ is useful to get an overview over what you can do