- 98-026 Nintendo. January 28, 2004
- Image Scaling
  - When playing games in an emulator, you usually play them at larger than the native size, so that they fill more than a tiny window. Some emulators offer advanced image scaling techniques that try to make the game look better when it's resized.
  - Nearest Neighbor is the simplest technique, which results in a blocky look. The destination image uses the physically closest corresponding pixel from the unscaled source image. This works pretty well for 2x or 3x scaling, but any scale that is not an exact multiple of the original will look terrible. Some of the first LCD computer monitors, especially the cheap ones, have the same problem.
  - Bilinear Filtering is often used in photo work and works well on images with natural color gradients. Most of the pixels in your destination do not correspond exactly to a pixel on the source; rather, they will correspond to a point somewhere in between source pixels. This method takes a weighted average of the (usually 4) actual source pixels surrounding that point and creates a new color for the destination. Note that this often results in blurriness.
  - Horizontal 50% is a technique made to simulate older televisions, where one field of scanlines will be brighter than the other. It usually uses nearest neighbor, but every second line is darker than normal. This works well for 2x scaling, except that it often looks dumb no matter what.
  - "Simple Smoothing", as RockNES calls it, is probably the same technique as Eagle, created by Dirk Stevens. The basic idea of this is that a "guessed" pixel (which does not correspond exactly to a source pixel) tries to set itself to the color of some set of 3 surrounding pixels.
    http://web.archive.org/web/20001031170950/www.retrofx.com/rfxeagleprinciple.html
  - Super Eagle, 2xSaI (Scaling and Interpolation), and Super 2xSaI are all effects created by Kreed. They are improvements on the basic idea of Eagle, and they tend to result in a cartoon-like image. They work best at 2x.
    http://elektron.its.tudelft.nl/~dalikifa/
  - hq3x is created for images 3 times as large as the original and has the same basic goals as some of the the other image scaling routines.
    http://www.hiend3d.com/hq3x.html
- How a Game Works
  - The basic concept of a video game has not changed much since they first appeared. This concept also applies to many other types of software, such as operating systems.
  - Initialization is the first step of the game. This makes sure the hardware is in a desirable state, all variables are set to their proper initial values, whatever those may be for the game, and the proper graphics are loaded in the pattern tables.
  - After initialization, the game loop is meant to run essentially forever, until the power is turned off, although some games may have more than one game loop and switch between them in certain situations.
  - Inside the game loop, the program keeps track of timing (30 frames per second), draws the screen with the most recent concept of the "world", gets player input from the joysticks, and performs calculations based on the input and the current state to determine the state of the game for the next frame. These calculations may include some amount of physics, calculating points, killing things, and playing the next frame of music.
- NES Registers

- The registers, sometimes also called "ports", are locations in memory that are hard-wired to certain functions. Your game can write to or read from these memory locations in order to perform certain system functions (displaying a sprite, changing a music note) or get information about the system (current joystick buttons, vertical blanking status)

- Joystick Input
  - The NES game pads are often referred to by developers as joysticks, simply for historical reasons, since previous game systems in general had joysticks instead of the directional pads (D-Pad) the NES had.
  - The joysticks on the NES are serial devices. This means that, instead of simply getting a single piece of data that tells the current state, which may change at any time, the game and the joysticks communicate with each other. The game sends a "strobe" signal, which forces the joystick to save its current button state and get ready to talk back. Then the game may read data from the joystick. The next 8 bytes read will give the status of the 8 buttons on the joystick. This concept also applies to the Power Pad input device, which has more than 8 buttons.
  - To strobe the joystick, send two bytes to the proper port. (player 1 is port $4016, and player 2 is port $4017). The first strobe byte is 1, and the second strobe byte is 0
  - The next 8 bytes read will tell the status of the buttons: A, B, Select, Start, Up, Down, Left, Right. Bit 0 tells whether the button was held down at the time of the strobe.
  - In some cases, the joystick data might have other bits set than bit 0, regardless of whether the button is pressed or not. This seems to be an indication of whether the joystick is plugged in or not. You should probably bitwise AND your data bytes with 1, so that you are dealing with strictly 1 or 0. The joystick routines I provide already do this.
- "A" Demo Rom
  - Available with source code from the Resources section of the webpage
  - Use pad to move sprite around screen
  - A and B change the sprite tile
  - Start returns the sprite to the center
- Accessing PPU Memory
  - The PPU has a different memory address space than the CPU. You can only manually access it through registers.
  - Write the high byte of your intended starting address to $2006, then write the low byte, also to $2006
  - Use $2007 to read and write data in PPU memory. The target address will increment automagically with each read or write
- Drawing Sprites
  - two methods: slower manual, and faster DMA
  - manual method
    - Sprite memory is 256 bytes. 4 bytes per sprite, therefore the 64 on-screen sprite limit. The first byte of each sprite is the Y coordinate of the sprite, next the tile number, next the attribute byte, and finally the X coordinate. The XY coordinates are for the upper-left corner of the sprite, measured from the upper-left corner of the screen.
    - The attribute byte is described in the slides.
  - DMA
    - Your game must keep a copy of sprite ram (256 byte array), aligned by $100. It is in the format described above.

- during the redraw period, write the high byte to $2003
- example: if your sprite memory copy is at address $200, then you can DMA the sprites by writing the value 2 to $2003

- Background Sprite Demo Rom
  - Available with source code from the Resources section of the webpage
  - Uses the standard method of sprite display (only 4 sprites to make SOF, no other sprites on screen)
  - Use pad to move SOF around the screen
  - Select will toggle displaying SOF behind the background
  - Start will toggle simple gravity (Left and Right to move, A to jump)
- Setting the Palette
  - The background and foreground palettes are in PPU memory, one after the other. You can write both palettes by telling the PPU to start writing at $3F00 (the background starting address) and then writing 32 bytes (16 per palette). This should be done during the screen retrace period.
- Scrolling
  - Scrolling is very simple to do. After the retrace period (and at any time during drawing, for advanced applications), write the horizontal scroll value, and then the vertical scroll value, to $2005. Each may be in the range of 0 to 255. Larger scroll values move the virtual screen window right and down, meaning that the background appears to move left and up.
  - More about scrolling will be covered in a later lecture
- Assignment 2
  - If you're going to drop the course, do it before you screw over several other people
  - Useful development tool
    - You must get this approved by me before you may use it for your assignment
    - Should be useful to other people
    - Should be usable to other people
    - Should run cross platform without major installation issues (consider wxWindows, SDL, GLUT, or FLTK if you are making a GUI)
    - Public Domain or other open source license
  - Complete game
    - Ideally, design a small game with good replay value. No need to kill yourself with a huge platformer (unless you want).
    - An artist on your team would be VERY useful
      - Take a look at box art from real NES games, design a front and back box cover
      - Check real NES manuals if you have them
      - posters are optional
  - Partial game
    - Assignments 2 and 3 become stages of the same project
      - Same team
      - Same game
    - The same guidelines apply as the Complete game, except more.
    - See me about what you plan to accomplish for the first part