

Prompt:

I need you to analyze a series of pictures regarding an exhibit that was at OMSI. Specifically, I want you to identify important parts of the exhibit immediately visible in these images, as well as the components relevant to this exhibit.



Result:

Based on the image you sent, here's an analysis of the visible important parts and relevant components of the OMSI exhibit:

Immediately Visible Important Parts:

- **"BUILD A COMPUTER" Sign:** This is the most prominent element and clearly

indicates the theme and purpose of this section of the exhibit. It suggests an interactive experience focused on assembling or understanding computers.

- **"citybop" Logo:** This likely represents the name of the specific exhibit or a sponsor. Its placement centrally suggests it's a key identifier.
- **Display Tables:** There are tables set up to showcase various components and interactive elements related to building a computer.
- **Clear Enclosure with Components:** On the left table, there's a transparent box containing various computer hardware components. These are likely examples of what one would use to build a computer.
- **Interactive Console/Device:** In the center, there's a black box with a screen and buttons. This appears to be an interactive element where visitors can engage with the process of building or learning about computers.
- **Panel with Components and Labels:** On the right table, there's a panel displaying different computer components (like memory sticks, expansion cards) with labels such as "CPU" and "ROM". This provides visual information about the different parts.
- **Colorful Backdrops:** The exhibit utilizes bright yellow, white, and magenta curved panels as a backdrop, making it visually engaging and drawing attention to the area.

Relevant Components Visible:

Based on what can be seen, the exhibit likely involves the following components:

- **Central Processing Unit (CPU):** A label "CPU" is visible on the right-hand panel.
- **Read-Only Memory (ROM):** A label "ROM" is visible on the right-hand panel.
- **Other Internal Components:** The clear enclosure likely contains other essential parts of a computer, though they are not clearly identifiable in this single image. These could include:
 - Motherboard
 - RAM (Random Access Memory)
 - Graphics Card
 - Power Supply
 - Storage devices (like hard drives or SSDs)
- **Interactive Screen and Controls:** The "citybop" labeled console suggests a digital interface for learning or simulating computer building.
- **Physical Examples of Components:** The panel on the right provides tangible examples of different computer parts.

In summary, the visible parts of the exhibit strongly suggest an interactive

learning experience at OMSI focused on demystifying the process of building a computer by showcasing its components and potentially allowing visitors to engage in a simulated assembly or learning activity.

If you provide more images, I can offer a more detailed and comprehensive analysis of the exhibit.

Prompt:

You are correct in that this is an interactive exhibit for building a computer. I am trying to figure out the specifics behind the interactive parts in the exhibit, but do not have a more clear understanding of what is included. Here is a closer visual of the image from before, focusing on the most important parts of the "build a computer" exhibit. Please ignore the "citybop" logo, as it is likely a watermark.

Here is a summary of what I understand from the image:

The screen displays relevant information, such as warning, error, or success results.

The blue button below the screen appears to have the label "System Check", indicating that it checks the interactive elements and goes through a series of evaluations to determine if an error, warning, or success message is displayed.

The square box appears to hold the majority of interactive objects.

The power supply appears to be the only object that is not interactive.



Result:

Based on the closer view, your understanding of the interactive parts of the OMSI "Build a Computer" exhibit seems quite accurate. Here's a more detailed breakdown and further observations:

Interactive Console (Black Box with Screen and Blue Button):

- **Screen Function:** As you correctly identified, the screen likely serves as the primary feedback mechanism for the interactive elements. It would display instructions, prompts, and the results of the "System Check" (error, warning, or success messages).
- **"System Check" Button:** The label on the blue button strongly suggests its role in evaluating the state of the interactive components. Pressing this button would likely trigger a process to determine if the virtual computer has been "built" correctly based on the user's interactions with the square box.

Square Box (Holding Interactive Objects):

- **Central Interactive Hub:** This box appears to be the core of the hands-on experience. It likely contains slots, connectors, or designated areas where visitors can physically interact with representations of different computer components.
- **Variety of Interactive Objects:** Given that the power supply is noted as the only non-interactive object, we can infer that the other items within or associated with this box are meant to be manipulated or placed in specific ways. These likely represent key computer components needed for a functional system.

Panel with Physical Components (Right Side):

- **Visual Reference:** This panel provides visual examples of real computer components like the CPU and ROM. While these specific components on the panel might not be directly manipulated as part of the *interactive building* process, they serve as important visual aids to help users understand what the interactive representations in the square box symbolize.
- **Potential for Matching:** It's possible that the interactive elements in the square box are designed to resemble these physical components, requiring users to identify and virtually install the correct parts.

Power Supply (Likely Non-Interactive):

- **Essential but Perhaps Abstracted:** The fact that the power supply is likely

non-interactive suggests that the exhibit might focus on the core logic and component relationships rather than the physical act of connecting power. It's a crucial component, but its role might be represented abstractly within the interactive system.

In summary, the interactive experience likely involves:

1. **Identifying and manipulating representations of computer components** within the square box.
2. **Placing these components in the correct locations or connecting them in the right way.**
3. **Using the "System Check" button to evaluate if the virtual computer has been assembled correctly.**
4. **Receiving feedback on the screen** in the form of error, warning, or success messages based on the "System Check."
5. **Using the physical component panel as a visual reference** to understand what the interactive elements represent.

Without further visuals or descriptions of the objects within the square box, it's difficult to pinpoint the exact nature of each interactive element. However, your deductions provide a solid understanding of the core mechanics of this "Build a Computer" exhibit at OMSI.

Prompt:

Based on observational data, I have determined that the unmarked components could belong to the following list of information:

CPU

Motherboard

GPU

Random Access Memory (volatile memory)

Computer Data Storage (long term memory)

Sound Card

Network Card

Would this be accurate information if the target audience of the exhibit are younger than 14 years old?

Result:

Yes, based on the typical components of a desktop computer and the educational nature of science museum exhibits aimed at younger audiences (under 14), your list of unmarked components is **highly likely to be accurate**.

Here's why:

- **Core Computer Components:** The listed items (CPU, Motherboard, GPU, RAM, Storage) represent the fundamental building blocks of a modern computer and are essential for understanding its basic functionality.
- **Common Peripherals/Expansion Cards:** Sound and Network cards are common examples of expansion cards that add important functionalities to a computer. Including these helps illustrate the modularity of computer systems.
- **Age Appropriateness:** For a younger audience, the exhibit would likely focus on the main, conceptually understandable components. While more technical details might be omitted, these listed items are generally introduced at a basic level in early science and technology education.
- **Interactive Potential:** Each of these components can be represented visually and interactively in a way that is engaging for children. For example, they could involve:
 - **Specific shapes or colors** for easy identification.
 - **Designated slots** on a model motherboard for placement.
 - **Simple visual representations** of their function (e.g., memory chips for RAM, a chip with fan for CPU).
- **Building a Mental Model:** Introducing these key components helps children build a foundational mental model of how a computer is constructed and how its different parts contribute to its overall operation.

Likely Simplifications:

Keep in mind that for this age group, the exhibit would likely involve simplifications:

- **Representational Objects:** The interactive components would likely be stylized representations rather than exact replicas of real hardware.
- **Simplified Connections:** The way components "connect" might be simplified for ease of use.
- **Focus on Function:** The emphasis would likely be on the basic function of

each component rather than intricate technical specifications.

In conclusion, your list of CPU, Motherboard, GPU, RAM, Computer Data Storage, Sound Card, and Network Card is a very reasonable and probable set of unmarked interactive components for a "Build a Computer" exhibit targeted at children younger than 14 years old. It covers the essential elements of a computer system in a way that is likely to be both educational and engaging for that age group.

Prompt:

Let's say this exhibit targets people between 6 and 14 years old. How would you explain each component, why adding them helps with building the computer, and the problems that arise when removing them?

Result:

Okay, let's break down how to explain each of those computer components to kids aged 6-14 in the context of this interactive exhibit, focusing on why they're important and what happens if they're missing:

1. CPU (Central Processing Unit) - The Brain:

- **Explanation (6-10 years):** "Imagine the CPU is like the brain of the computer. It's the super smart part that does all the thinking and follows instructions. When you click on something or tell the computer to do something, the CPU figures out how to make it happen."
- **Explanation (11-14 years):** "The CPU is the central processing unit, and it's the core of the computer's logic. It executes instructions from software, performs calculations, and manages all the other hardware components. Think of it as the conductor of an orchestra, telling everyone what to do."
- **Why it's needed:** "Without a brain, you can't think or do anything, right? It's the same with the computer. Without the CPU, the computer can't run any programs, open files, or do anything at all. It's just a bunch of parts that don't know what to do."
- **Problem if removed:** "If you take out the CPU, the computer won't turn on or do anything. It's like trying to have a body without a brain – it just won't work."

2. Motherboard - The Body's Nervous System:

- **Explanation (6-10 years):** "The motherboard is like the body of the computer

and all the roads and connections inside your body. It has special spots for all the other parts to plug into, like the brain, memory, and graphics card. It helps them all talk to each other."

- **Explanation (11-14 years):** "The motherboard is the main circuit board in the computer. It provides the electrical connections and communication pathways between all the other components, including the CPU, RAM, GPU, and storage. It's the foundation that everything else builds upon."
- **Why it's needed:** "Think of it like trying to build with LEGOs without a baseplate. You have all the pieces, but no way to connect them properly. The motherboard lets all the different parts of the computer connect and share information."
- **Problem if removed:** "If you don't have a motherboard, there's nowhere for the other parts to plug in and communicate. Even if you have a brain (CPU) and memory, they can't work together without the motherboard connecting them."

3. GPU (Graphics Processing Unit) - The Picture Maker:

- **Explanation (6-10 years):** "The GPU is like the artist of the computer. It's really good at drawing pictures, videos, and games on the screen. It makes everything look colorful and move smoothly."
- **Explanation (11-14 years):** "The GPU is a specialized processor designed to handle graphics rendering. It takes data and transforms it into the images you see on your monitor, which is especially important for games, videos, and complex visual applications."
- **Why it's needed:** "While the CPU can do some basic picture stuff, the GPU is much faster and better at it. If you want to play games or watch videos without them being slow and choppy, you need a good GPU."
- **Problem if removed:** "If you don't have a GPU (or if it's not working well), the pictures and videos on the screen might look very basic, slow, or even not show up at all. Games would be very difficult or impossible to play."

4. Random Access Memory (RAM) - Short-Term Memory:

- **Explanation (6-10 years):** "Imagine RAM is like the computer's short-term memory, like what you remember right now. When you open a game or a website, the computer puts the important stuff it needs right away into the RAM so it can get to it quickly. The more RAM you have, the more things the computer can remember at once without slowing down."
- **Explanation (11-14 years):** "RAM is volatile memory that the computer uses to store data that it's actively working on. When you open an application or file,

it's loaded into RAM for quick access by the CPU. The more RAM you have, the more applications and data the computer can handle simultaneously without needing to constantly access slower storage."

- **Why it's needed:** "Without RAM, the computer would have to constantly go back to its long-term storage (like a slow filing cabinet) every time it needed information. This would make everything super slow and frustrating."
- **Problem if removed:** "If you don't have RAM, the computer won't be able to run programs or even start up properly. It needs that short-term memory to hold the instructions it's currently using."

5. Computer Data Storage (Long-Term Memory) - The Filing Cabinet:

- **Explanation (6-10 years):** "This is like the computer's long-term memory, like your notebooks and photo albums. It's where all your files, games, pictures, and videos are saved even when the computer is turned off. We have different types like hard drives or faster SSDs."
- **Explanation (11-14 years):** "Computer data storage, such as Hard Disk Drives (HDDs) or Solid State Drives (SSDs), is non-volatile memory used to permanently store the operating system, applications, and user data. This allows the computer to retain information even when powered off."
- **Why it's needed:** "Without storage, anything you do on the computer would disappear when you turn it off. You wouldn't be able to save your drawings, games, or homework."
- **Problem if removed:** "If you take out the storage, the computer won't have any place to save its important information, including the instructions to even start up. It would be like having a brain and memory but nowhere to keep your books and important papers."

6. Sound Card - The Speaker's Voice:

- **Explanation (6-10 years):** "The sound card is what lets the computer make sounds! It takes the digital sounds from games and music and turns them into sounds you can hear through speakers or headphones."
- **Explanation (11-14 years):** "The sound card processes audio signals, converting digital data into analog signals that can be outputted through speakers or headphones. It also handles audio input from microphones."
- **Why it's needed:** "If you want to hear music, sound effects in games, or people talking on a video call, you need a sound card to make those sounds come out."
- **Problem if removed:** "Without a sound card, the computer won't be able to play any sounds. Games will be silent, and you won't hear any audio from

videos or music."

7. Network Card - The Connection to the World:

- **Explanation (6-10 years):** "The network card is like the computer's way of talking to other computers and the internet! It helps you connect to Wi-Fi or plug in a cable so you can visit websites, play online games, and talk to friends."
- **Explanation (11-14 years):** "The network card (or network interface card - NIC) allows the computer to connect to a network, either wired (Ethernet) or wirelessly (Wi-Fi). This enables communication with other devices and access to the internet."
- **Why it's needed:** "If you want to go online, visit websites, send emails, or play games with people far away, you need a network card to connect to the internet."
- **Problem if removed:** "Without a network card, the computer won't be able to connect to the internet or other computers on a network. You won't be able to browse websites or play online games."

By using these analogies and simple explanations, you can effectively teach children in the 6-14 age range about the essential components of a computer and why they are crucial for it to function. The interactive exhibit provides a hands-on way for them to solidify these concepts.

Prompt:

What computer components could have more than one member in its group based on the analysis of the image? The image clearly details some dark colored interactive objects are stationed next to each other while others are separated by position, which could indicate groups of one type of component (for example, RAM). Are there some components in a computer other than RAM or the ones already mentioned that could apply? Is it possible to have multiple processors?

Please organize the results into groups based on how likely there is more than one of these.

Result:

Based on the image and general computer architecture, here's an analysis of computer components that could potentially have more than one member in their group within the exhibit, organized by likelihood:

Highly Likely to Have More Than One:

- **Random Access Memory (RAM):** This is the most likely component to be represented by multiple physical modules. Modern computers almost always use multiple RAM sticks working together to increase memory bandwidth and capacity. The description of dark colored interactive objects stationed next to each other strongly suggests this.

Possible to Have More Than One (But Less Common in Typical Consumer PCs):

- **Computer Data Storage (Long-Term Memory):** While a typical consumer PC has one primary storage drive (SSD or HDD), it's very common to have multiple storage devices for increased capacity or organization (e.g., a smaller, fast SSD for the operating system and a larger HDD for files, or multiple identical drives in a RAID configuration for redundancy or performance). The exhibit might represent this possibility.
- **Graphics Processing Unit (GPU):** While most consumer PCs have a single dedicated GPU, some high-end systems, particularly those for gaming or professional graphics work, can have multiple GPUs (though this is becoming less common with modern integrated solutions). It's less likely for an exhibit aimed at a younger audience to focus on multi-GPU setups due to complexity.
- **Network Card:** A standard computer usually has one primary network interface (either wired Ethernet or wireless Wi-Fi integrated into the motherboard). However, it's possible to have additional network cards for specific purposes, such as having both a wired and a wireless card active simultaneously or having multiple Ethernet ports for network bridging or advanced networking setups. This is less common for the target age group but not impossible to represent.

Less Likely to Have More Than One in a Standard Consumer PC Context:

- **Central Processing Unit (CPU):** Modern consumer desktop computers almost universally have a single CPU socket and therefore a single main processor. While multi-processor systems exist (primarily in servers and high-end workstations), they are significantly more complex and less relevant for an introductory exhibit aimed at younger children. The exhibit would almost certainly focus on the fundamental concept of a brain for the computer.
- **Sound Card:** While some high-end audio setups might involve multiple sound cards for specific professional audio tasks, the vast majority of consumer PCs have a single integrated or dedicated sound card. It's unlikely an introductory exhibit would feature multiple sound cards.
- **Motherboard:** A computer has only one motherboard, as it's the foundational

circuit board that connects all other components.

Is it possible to have multiple processors?

Yes, it is absolutely possible to have multiple processors in a single computer system. These are known as **multi-processor systems**. However, as mentioned above, this is much more common in:

- **Servers:** Designed to handle heavy workloads and many users simultaneously.
- **High-End Workstations:** Used for demanding tasks like video editing, 3D rendering, and scientific simulations.

Consumer desktop computers for everyday use and gaming almost always utilize a single CPU socket on the motherboard, even if that CPU has multiple *cores* (which are individual processing units within a single physical chip).

In the context of the exhibit for a younger audience, it's most probable that the multiple dark-colored interactive objects represent RAM modules. The possibility of multiple storage devices is also plausible. Multiple GPUs or network cards are less likely but not entirely out of the question depending on the level of detail the exhibit aims to convey. Multiple CPUs or motherboards would be highly unlikely for this target audience.

Prompt:

Lets start at the beginning. For now assume that that the power unit is already installed, the motherboard is already represented, and that three of the interactive components are likely "ram". For an audience of ages 6 to 14 years, what are the top ten components that would be in a computer given that the number of interactive components being eleven? How would you explain this to the target audience? What would happen when each one is added or removed?

Result:

Okay, let's break down the top ten components for an eleven-component interactive "Build a Computer" exhibit, assuming power and motherboard are pre-installed, and three RAM slots are present. We'll aim for clarity and age-appropriate language for 6-14 year olds:

Top Ten Components (Given Eleven Total, Three RAM):

1. **CPU (Central Processing Unit)**

2. **RAM (Random Access Memory) - 3 slots**
3. **GPU (Graphics Processing Unit)**
4. **Storage Drive (SSD or HDD)**
5. **Sound Card**
6. **Network Card (Ethernet or Wi-Fi)**
7. **Cooling System (Heatsink/Fan)**
8. **Case Fan (Additional Cooling)**
9. **Input Device Connector (e.g., USB port representation)**
10. **Output Device Connector (e.g., HDMI port representation)**

Explanations for the Target Audience (6-14 years):

1. CPU (Central Processing Unit):

- **Explanation:** "This is the computer's brain! It does all the thinking and follows the instructions for everything you do on the computer."
- **Adding:** "If you put the CPU in, the computer can start to process information and follow commands."
- **Removing:** "Without the CPU, the computer is like a toy without batteries – it won't do anything."

2. RAM (Random Access Memory) - 3 slots:

- **Explanation:** "This is the computer's short-term memory. It remembers things it needs to use right now, like when you're playing a game or watching a video. The more RAM you have, the more the computer can remember at once, making it faster."
- **Adding:** "Putting in RAM lets the computer run more things at the same time without slowing down."
- **Removing:** "Without RAM, the computer will be very slow or won't work at all because it can't remember what it's doing."

3. GPU (Graphics Processing Unit):

- **Explanation:** "This is the part that makes the pictures and videos look good. It's like the computer's artist! It helps the computer show games and movies smoothly."
- **Adding:** "Putting in the GPU makes the pictures and videos on the screen look clear and smooth."
- **Removing:** "Without the GPU, pictures and videos might look blurry or choppy, and games might not work well."

4. **Storage Drive (SSD or HDD):**

- **Explanation:** "This is like the computer's filing cabinet. It's where it stores all your files, games, and programs even when you turn it off. We have different kinds like fast SSDs or big HDDs."
- **Adding:** "Putting in the storage drive lets you save all your work and games so you can use them again later."
- **Removing:** "Without the storage drive, you can't save anything, and the computer won't remember anything when you turn it off."

5. **Sound Card:**

- **Explanation:** "This is what lets the computer make sounds! It helps you hear music, game sounds, and people talking in videos."
- **Adding:** "Putting in the sound card lets you hear all the sounds the computer makes."
- **Removing:** "Without the sound card, the computer will be silent – you won't hear any music or sounds."

6. **Network Card (Ethernet or Wi-Fi):**

- **Explanation:** "This is how the computer connects to the internet! It lets you visit websites, play online games, and talk to friends far away."
- **Adding:** "Putting in the network card lets you connect to the internet and talk to other computers."
- **Removing:** "Without the network card, you can't go online or play games with friends on the internet."

7. **Cooling System (Heatsink/Fan):**

- **Explanation:** "This helps keep the CPU cool. The CPU gets hot when it works hard, so we need a fan to blow air and a heatsink to take away the heat."
- **Adding:** "Putting in the cooling system keeps the CPU from getting too hot and breaking."
- **Removing:** "Without the cooling system, the CPU could get too hot and stop working."

8. **Case Fan (Additional Cooling):**

- **Explanation:** "This is another fan that helps keep the whole computer cool. It moves air around inside the computer to keep all the parts from getting too hot."

- **Adding:** "Putting in the case fan helps keep all the parts cool and working well."
- **Removing:** "Without the case fan, the inside of the computer could get too hot, and some parts might stop working."

9. Input Device Connector (e.g., USB port representation):

- **Explanation:** "This is how you connect things like a keyboard or mouse to the computer. It lets you tell the computer what to do."
- **Adding:** "Putting in the input connector lets you plug in a keyboard or mouse and control the computer."
- **Removing:** "Without the input connector, you can't use a keyboard or mouse to tell the computer what to do."

10. Output Device Connector (e.g., HDMI port representation):

- **Explanation:** "This is how you connect a monitor or TV to the computer. It lets you see what the computer is doing."
- **Adding:** "Putting in the output connector lets you see the pictures and videos the computer makes on a screen."
- **Removing:** "Without the output connector, you won't be able to see anything the computer is doing on a screen."

This breakdown provides a clear and engaging way to teach children about the essential components of a computer and the impact of adding or removing them in the context of the interactive exhibit.

Prompt:

Can you analyze this code and provide proper comments throughout to explain its current behavior?

```
// get elements from the HTML document using their IDs or selectors
```

```
const canvas = document.querySelector("#drag-canvas");
```

```
const context = canvas.getContext("2d");
```

```
/*
```

```
const coordDisplay = document.querySelector("#coordinates");
```

```
const overlapDisplay = document.querySelector("#isOver");

*/

let currentlyDragging = null;

let dragOffsetX = 0;

let dragOffsetY = 0;

let isDragging = false;


// Class definition for creating the monitor viewport

class Monitor {

    // a monitor needs x, y, width, height, and a highlight color

    constructor (positionX, positionY, width = 100, height = 200, highlightColor =
"orange") {

        this.posX = positionX;

        this.posY = positionY;

        this.width = width;

        this.height = height;

        this.highlight = highlightColor;

        this.text = "";

        this.description = "";

        this.buffer = 100;

        this.isHover = false;

    }

}
```

```
// set text and description

setMonitorValues(text, description) {

  this.text = text;

  this.description = description;

}
```

```
/* Solution courtesy of
https://stackoverflow.com/questions/2936112/text-wrap-in-a-canvas-element */
```

```
getLines(ctx, text, maxWidth) {

  let words = text.split(" ");

  let lines = [];

  let currentLine = words[0];

  for (let i = 1; i < words.length; i++) {

    let word = words[i];

    let width = ctx.measureText(currentLine + " " + word).width;

    if (width < maxWidth) {

      currentLine += " " + word;

    } else {

      lines.push(currentLine);

      currentLine = word;

    }

  }

}
```

```
    }  
  }  
  lines.push(currentLine);  
  return lines;  
}
```

```
// draw the entire monitor, starting with outer board and going to inner board
```

```
drawBoard(ctx, text) {  
  this.drawBoardOuter(ctx);  
  this.drawBoardInner(ctx, text);  
}
```

```
// draw the inner rectangle for the monitor
```

```
drawBoardInner(ctx, title = null, text = null, fillColor = "") {  
  // set a buffer for the inner rectangle of the monitor  
  let buffer = 15;  
  let innerX = this.posX + buffer;  
  let innerY = this.posY + buffer;  
  let innerWidth = this.width - (2*buffer);  
  let textX = innerX + 10;  
  let textY = innerY + 25;  
  // inner rectangle
```

```
if(fillColor) ctx.fillStyle = fillColor;

else ctx.fillStyle = "black";

ctx.strokeRect(innerX, innerY, innerWidth, this.height - (2*buffer));

ctx.fillRect(innerX, innerY, innerWidth, this.height - (2*buffer));

ctx.strokeRect(this.posX, this.posY + this.height, this.width, this.buffer);

// initialize variables for generating text

let textFont = 20;

let defaultText = "Add or remove a component to get started! Once you are done,
you can click the system check button to see if your computer works!";

let formattedText = null;

let screenLength = innerWidth;


// set the ctx fill style and font

ctx.fillStyle = "lime";

ctx.font = `${textFont}px Arial`;


// add title if exists

if(title != null) {

    textFont = 36;

    textY += 20;

    ctx.font = `${textFont}px Arial`;

    ctx.fillText(title, textX, textY, screenLength);

    textY += textFont;
```

```

}

// set the ctx fill style and font for regular text
textFont = 20;
ctx.fillStyle = "lime";
ctx.font = `${textFont}px Arial`;

// display the normal text field if exists. Else, display default text.
if (text != null && text.isArray == false) {
    formattedText = this.getLines(ctx, text, screenLength);
} else {
    formattedText = this.getLines(ctx, defaultText, screenLength);
}

for ( i = 0; i < formattedText.length; i++ ) {
    if(formattedText.length == 1) {
        ctx.fillText(formattedText[i], textX, textY + textFont*i, screenLength);
    } else {
        ctx.fillText(formattedText[i], textX, textY + textFont*i, screenLength);
    }
}
}

```



```
// draw the outside of the monitor

drawBoardOuter(ctx) {

    // outer rectangle

    ctx.strokeStyle = "black";

    ctx.fillStyle = "tan";

    ctx.fillRect(this.posX, this.posY, this.width, this.height);

    ctx.strokeRect(this.posX, this.posY, this.width, this.height);

}
```

```
// draw the system check button

drawButton(ctx) {

    let circleX = this.posX + this.width * 2 / 3;

    let circleY = this.posY + this.height + this.buffer/2;

    let radius = (this.buffer / 2) - 10;

    ctx.fillStyle = this.highlight;

    ctx.beginPath();

    ctx.arc(circleX, circleY, radius, 0, 2 * Math.PI);

    ctx.fill();

    ctx.closePath();

    ctx.stroke();

}
```

```
// draw the highlights for the system check button

drawHighlight(ctx) {
  if(this.isHover) {
    // variables to hold starting positions

    let startX1 = this.posX + this.width * 2 / 3 - this.buffer / 2;
    let startY1 = this.posY + this.height + this.buffer / 2;
    let startX2 = this.posX + this.width * 2 / 3 + this.buffer / 2;
    let diagY = 5;

    ctx.strokeStyle = this.highlight;

    // draw the first line
    ctx.beginPath();
    ctx.fillStyle = "";
    ctx.strokeStyle = this.highlight;
    ctx.moveTo(startX1, startY1);
    ctx.lineTo(startX1 - 20, startY1);
    ctx.closePath();
    ctx.stroke();

    // draw the second line
```

```
ctx.beginPath();  
ctx.fillStyle = "";  
ctx.strokeStyle = this.highlight;  
ctx.moveTo(startX1, startY1 + (4*diagY));  
ctx.lineTo(startX1 - 20, startY1 + (6*diagY));  
ctx.closePath();  
ctx.stroke();
```

// draw the third line

```
ctx.beginPath();  
ctx.fillStyle = "";  
ctx.strokeStyle = this.highlight;  
ctx.moveTo(startX1, startY1 - (4*diagY));  
ctx.lineTo(startX1 - 20, startY1 - (6*diagY));  
ctx.closePath();  
ctx.stroke();
```

// draw the first line on the other side

```
ctx.beginPath();  
ctx.fillStyle = "";  
ctx.strokeStyle = this.highlight;  
ctx.moveTo(startX2, startY1);
```

```
ctx.lineTo(startX2 + 20, startY1);
```

```
ctx.closePath();
```

```
ctx.stroke();
```

```
// draw the second opposite line
```

```
ctx.beginPath();
```

```
ctx.fillStyle = "";
```

```
ctx.strokeStyle = this.highlight;
```

```
ctx.moveTo(startX2, startY1 + (4*diagY));
```

```
ctx.lineTo(startX2 + 20, startY1 + (6*diagY));
```

```
ctx.closePath();
```

```
ctx.stroke();
```

```
// draw the third opposite line
```

```
ctx.beginPath();
```

```
ctx.fillStyle = "";
```

```
ctx.strokeStyle = this.highlight;
```

```
ctx.moveTo(startX2, startY1 - (4*diagY));
```

```
ctx.lineTo(startX2 + 20, startY1 - (6*diagY));
```

```
ctx.closePath();
```

```
ctx.stroke();
```

```
}
```

```
}
```

```
drawButtonLabel(ctx) {  
    // variables for x and y of text  
    let textX = this.posX + this.buffer / 2;  
    let textY = this.posY + this.height + this.buffer / 2 + 10;  
    // change stroke color and stroke style  
    ctx.strokeStyle = "black";  
    ctx.fillStyle = "black";  
    // add the text in 30px Arial;  
    ctx.font = "30px Arial";  
    ctx.fillText("System Check", textX, textY);  
}
```

```
drawArrowToButton(ctx) {  
    // variables to determine start of arrow  
    let startX = this.posX + this.width / 2 - 20;  
    let startY = this.posY + this.height + this.buffer / 2 - 5;  
    // set colors to highlight color  
    ctx.strokeStyle = this.highlight;  
    ctx.fillStyle = this.highlight;
```

```
ctx.fillStyle = this.highlight;

// starting position
ctx.moveTo(startX, startY);

// start drawing shape
ctx.lineTo(startX + 20, startY);
ctx.lineTo(startX + 20, startY - 10);
ctx.lineTo(startX + 35, startY + 5);
ctx.lineTo(startX + 20, startY + 20);
ctx.lineTo(startX + 20, startY + 10);
ctx.lineTo(startX, startY + 10);
ctx.lineTo(startX, startY);
ctx.closePath();
ctx.fill();

// stroke the shape
ctx.stroke();
}
```

```
draw(ctx, text) {
  this.drawButtonLabel(ctx);
  this.drawArrowToButton(ctx);
  this.drawButton(ctx);
  this.drawHighlight(ctx);
}
```



```

    this.drawBoard(ctx, "Welcome!", text);
}

// Method to check if a given mouse coordinate is inside the bounds of the port
isOverButton(mouseX, mouseY) {
    // get circle position x and y
    let circleX = this.posX + this.width * 2 / 3;
    let circleY = this.posY + this.height + this.buffer/2;
    // get circle radius
    let radius = (this.buffer / 2) - 10;
    // get distance x and y using mouse and circle position
    let distX = mouseX - circleX;
    let distY = mouseY - circleY;
    // calculate the hypotenuse via pythagorean theorem
    //  $(h*h) = (a*a) + (b*b)$  OR  $h = \text{square root of } ((a*a) + (b*b))$ 
    let h = Math.sqrt(Math.pow(distX, 2) + Math.pow(distY, 2));
    // return whether hypotenuse is smaller than circle radius
    return (
        h <= radius
    );
}
}

```

// Class definition for creating rectangle objects

```
class Rectangle {  
    constructor(width, height, posX, posY, fillColor, text="") {  
        this.width = width;  
        this.height = height;  
        this.currX = posX;  
        this.currY = posY;  
        this.fillColor = fillColor;  
        this.textValue = text;  
        this.originalX = posX; // Store original X position  
        this.originalY = posY; // Store original Y position  
        this.inPort = null; // Keep track of which port it is in, null if not in a port  
    }  
}
```

// Method to draw the rectangle on the canvas

```
draw(ctx) {  
    ctx.fillStyle = this.fillColor;  
    ctx.fillRect(this.currX, this.currY, this.width, this.height);  
    if (this.textValue) {  
        ctx.fillStyle = "black";  
        ctx.font = "12px Arial";
```

```
    ctx.fillText(this.textValue, this.currX + this.width/6, this.currY + this.height/2 + 5);  
  }  
}
```

```
// Method to reset the position of the rectangle to its original coordinates  
resetPosition() {  
    this.currX = this.originalX;  
    this.currY = this.originalY;  
    this.inPort = null; // Clear the inPort property when resetting  
}  
}
```

```
// Class definition for creating "Port" objects  
class Port {  
    // Removed currStrokeStyle, keep the style consistent  
    isActive = false;  
    occupiedShape = null; // Keep track of what's in the port  
    constructor(width, height, posX, posY, strokeColor) {  
        this.width = width;  
        this.height = height;  
        this.currX = posX;
```

```
this.currY = posY;

this.strokeColor = strokeColor; // This is the *fixed* stroke color
}

getPortShapeID() {
    if (this.occupiedShape != null) return this.occupiedShape.textValue;
    else return null;
}

// Method to draw the port on the canvas
draw(ctx) {
    ctx.strokeStyle = this.strokeColor; // Use the fixed stroke color
    ctx.strokeRect(this.currX, this.currY, this.width, this.height);
    this.drawIndicator(ctx);
}

// Method to draw a color indicator next to the port
drawIndicator(ctx) {
    let indicatorColor = 'red'; // Default color
    if (this.occupiedShape !== null)
    {
```

```
    indicatorColor = 'green';  
}  
  
if (this.isActive) {  
    indicatorColor = "yellow";  
}  
  
ctx.fillStyle = indicatorColor;  
  
ctx.fillRect(this.currX + this.width + 10, this.currY, 10, 10);  
}
```

// Method to check if a given mouse coordinate is inside the bounds of the port

```
isOver(mouseX, mouseY) {  
    return (  
        mouseX >= this.currX &&  
        mouseX <= this.currX + this.width &&  
        mouseY >= this.currY &&  
        mouseY <= this.currY + this.height  
    );  
}  
}
```

// Function to check if a given mouse coordinate is inside a given rectangle object

```
function isOverlap(mouseX, mouseY, rectangle) {
  return (
    mouseX >= rectangle.currX &&
    mouseX <= rectangle.currX + rectangle.width &&
    mouseY >= rectangle.currY &&
    mouseY <= rectangle.currY + rectangle.height
  );
}
```

```
////////////////////////////////////
```

```
/* Primary driver code block */
```

```
////////////////////////////////////
```

```
/*
```

Define constant for potential error/success states

Unless otherwise noted, most responses have been gathered using generative AI feedback */

```
const states = {
```

```
  "cpu": {
```

```
    "success": "We are now capable of reading instructions in our computer.",
```

```
    "error": "The CPU is the brain. We cannot read instructions without it."
```

```
  },
```


/* Details on rom found at <https://www.geeksforgeeks.org/read-only-memory-rom/> */

"rom": {

"success": "This is a recipe book that your computer cannot lose, storing important system instructions within.",

"error": "Without rom, the computer will never wake up. It holds important instructions on how to do so."

},

"ram": {

"success": "Ram helps the computer quickly remember items. Adding more can help the computer do jobs faster.",

"surplus": "Adding extra ram helps the computer do things faster!",

"warning": "Removing ram reduces a computer's ability to think and multitask. It will be slower when using more than one function.",

"error": "Without ram, the computer will not work. At least one is needed to help read the instructions on how to boot up."

},

"gpu": {

"success": "This is the computer's artist! It helps the computer make images, videos, and games look good.",

"error": "Without the GPU, pictures and videos might look blurry or choppy, and games might not work well."

},

"storage": {

"success": "Putting in the storage drive lets you save all your work and games so you can use them again later.",

"surplus1": "Adding more storage lets the computer remember more. Memory was

expensive for computers for a while, but got cheaper very fast.",

"surplus2": "Now the computer can remember a lot. This is probably how much current computers can remember.",

"warning": "Removing memory can cause the computer to remember less. The stuff on the removed component could still be seen.",

"error": "Without the storage drive, you can't save anything, and the computer won't remember anything when you turn it off."

},

"sound": {

"success": "Putting in the sound card lets you hear all the sounds the computer makes.",

"error": "Without the sound card, the computer will be silent – you won't hear any music or sounds."

},

"network": {

"success": "Putting in the network card lets you connect to the internet and talk to other computers.",

"error": "Without the network card, you can't go online or play games with friends on the internet."

},

"system": {

"success": "Success! Your computer is fully functional.",

"surplus": "Success! Not only does it work, your computer is better than before!",

"complete": "Wow! Your computer has all the best components available!",

"error": "Oops! Something is missing from the computer that is needed to function. Try adding more components."

```
},  
}
```

```
// Define monitor for the canvas
```

```
const monitor = new Monitor((canvas.width / 2) + 20, 60, (canvas.width / 2) - 40, 400,  
"Red");
```

```
// Define the initial properties for the rectangles
```

```
const shapes = [[80, 20, 100, 80, 3, "RAM"],  
                [80, 20, 40, 200, 1, "GPU"],  
                [80, 20, 200, 80, 3, "Storage"],  
                [80, 20, 40, 240, 1, "Network"],  
                [80, 20, 40, 280, 1, "Sound"],  
                [40, 40, 40, 80, 1, "CPU"],  
                [40, 40, 40, 140, 1, "ROM"]];
```

```
// Define the initial properties for the ports
```

```
const portShapes = [[80, 20, 340, 60, 3],  
                   [80, 20, 480, 60, 3],  
                   [80, 20, 340, 180, 3],  
                   [40, 40, 480, 180, 1],  
                   [40, 40, 480, 240, 1]];
```

```
// Define an array of colors to be used for components
```

```
const colors = ["pink", "lightblue", "lightgreen", "orange", "cyan", "yellow",  
"AntiqueWhite", "Beige"];
```

```
// define a variable to hold all known component types
```

```
let componentCount = {
```

```
  "ram": {
```

```
    "count": 0,
```

```
    "limit": 3,
```

```
  },
```

```
  "storage": {
```

```
    "count": 0,
```

```
    "limit": 3,
```

```
  },
```

```
  "gpu": {
```

```
    "count": 0,
```

```
    "limit": 1,
```

```
  },
```

```
  "network": {
```

```
    "count": 0,
```

```
    "limit": 1,
```

```
  },
```

```
  "sound": {
```

```
    "count": 0,
```

```
    "limit": 1,
```

```
},  
"cpu": {  
  "count": 0,  
  "limit": 1,  
},  
"rom": {  
  "count": 0,  
  "limit": 1,  
},  
}
```

```
// define some preliminary variables
```

```
let rectangles = [];
```

```
let ports = [];
```

```
let i = 0;
```

```
// Loop through the 'shapes' array to create Rectangle objects and add them to the  
'rectangles' array
```

```
for (i = 0; i < shapes.length; i++) {
```

```
  for (j = 0; j < shapes[i][4]; j++ ) {
```

```
    let rect = new Rectangle(shapes[i][0], shapes[i][1], shapes[i][2], shapes[i][3] + (j *  
40), colors[i % colors.length], shapes[i][5]);
```

```
    rectangles.push(rect);  
  }  
}
```

// Loop through the 'portShapes' array to create Port objects and add them to the 'ports' array

```
for (i = 0; i < portShapes.length; i++) {  
  for (j = 0; j < portShapes[i][4]; j++) {  
    let port = new Port(portShapes[i][0], portShapes[i][1], portShapes[i][2],  
      portShapes[i][3] + (j*40), "black");  
    ports.push(port);  
    // console.log(`${i}, ${j}: (i, j)`);  
  }  
}
```

// function to alter component count based on object and value

```
function componentModifier(obj, name, mod="") {  
  let comp = name.toLowerCase();  
  if (mod == "" || obj == null || name == "") return;  
  if (mod == "+" && (obj.comp.count + 1 <= obj.comp.limit)) {  
    obj.comp.count += 1;  
    return "pos";  
  }  
}
```

```
} else if (mod == "-" && (obj.comp.count + 1 <= obj.comp.limit)) {  
    obj.comp.count -= 1;  
    return "neg";  
} else {  
    return null;  
}  
}
```

// function to get object from rectangle name

```
function stateSwitch(state) {  
    // init variable to hold state object  
    let actualState = null;  
    let str = state.toLowerCase();  
  
    /* get the appropriate state based on the given entry value */  
    if(str == "ram") { // if ram get ram state  
        actualState = states.ram;  
    } else if (str == "storage") { // if storage get storage state  
        actualState = states.storage;  
    } else if (str == "cpu") { // if cpu get cpu state, etc.  
        actualState = states.cpu;  
    } else if (str == "rom") {
```

```
    actualState = states.rom;
} else if (str == "gpu") {
    actualState = states.gpu;
} else if (str == "network") {
    actualState = states.network;
} else if (str == "sound") {
    actualState = states.sound;
} else if (str == "system") {
    actualState = states.system;
}

// return the actualState; if no object, it returns null
return actualState;
}

// function to draw the monitor
function drawMonitor(textInput) {
    if(monitor != null) {
        monitor.draw(context, textInput);
    }
}
```



```
// function to draw the divider labels

function drawDividerLabels(ctx, width, height) {

    // variables for x and y of text

    let componentX = width * (1/16);

    let portX = width * (11/32);

    let monitorX = width * (21/32);

    let distY = 40;

    // change stroke color and stroke style

    ctx.strokeStyle = "black";

    ctx.fillStyle = "black";

    // add the text in 30px Arial;

    ctx.font = "30px Arial";

    ctx.fillText("Components", componentX, distY);

    ctx.fillText("Ports", portX, distY);

    ctx.fillText("System Monitor", monitorX, distY);

}
```

```
// function to draw the lines to divide the canvas

function drawDividers() {

    // create variables to hold canvas width, height, and buffer value
```

```
let w = canvas.width;

let h = canvas.height;

let buffer = 10;

// set context styles

context.fillStyle = "lightgrey";

context.strokeStyle = "lightgrey";

context.fillRect(0, 0, w, h);

// reset context styles and line width

context.fillStyle = "tan";

context.strokeStyle = "tan";

context.lineWidth = 10;

// draw a rectangle around the canvas

context.strokeRect(0, 0, w, h);

// draw a dividing line between components and ports

context.fillRect(w/4, buffer, 10, h - (buffer * 2));

// draw a dividing line between ports and monitor

context.fillRect(w/2, buffer, 10, h - (buffer * 2));

// reset line width

context.lineWidth = 1;

drawDividerLabels(context, w, h);

}
```

```
// Function to clear the canvas and redraw all rectangles and ports
```

```
function drawRectangles() {  
  ports.forEach(port => {  
    port.draw(context);  
  });  
  rectangles.forEach(rectangle => {  
    rectangle.draw(context);  
  });  
}
```

```
function drawCanvas(textInput) {  
  context.clearRect(0, 0, canvas.width, canvas.height); // Clear canvas before drawing  
  drawDividers();  
  drawMonitor(textInput);  
  drawRectangles();  
}
```

```
// Initial call to the drawRectangles function to draw the shapes and ports when the  
page loads
```

```
drawCanvas("");
```

```
// Function to update the coordinates displayed in the 'coordDisplay' element
```

```
function updateCoordinates(event) {  
    const rect = canvas.getBoundingClientRect();  
    const x = event.clientX - rect.left;  
    const y = event.clientY - rect.top;  
    coordDisplay.textContent = `X: ${Math.floor(x)}, Y: ${Math.floor(y)}`;  
}
```

// Attach an event listener to the canvas for the 'mousemove' event

```
canvas.addEventListener("mousemove", (e) => {
```

```
    // update coordinates
```

```
    // updateCoordinates(e);
```

```
    const rect = canvas.getBoundingClientRect();
```

```
    const mouseX = e.clientX - rect.left;
```

```
    const mouseY = e.clientY - rect.top;
```

```
    if (monitor.isOverButton(mouseX, mouseY)) {
```

```
        monitor.isHover = true;
```

```
    } else {
```

```
        monitor.isHover = false;
```

```
    }
```

```
if (!isDragging) {  
  ports.forEach(port => {  
    if (port.isOver(mouseX, mouseY)) {  
      port.isActive = true;  
    } else {  
      port.isActive = false;  
    }  
  });  
  drawCanvas();  
}
```

```
if (currentlyDragging && isDragging) {  
  currentlyDragging.currX = mouseX - dragOffsetX;  
  currentlyDragging.currY = mouseY - dragOffsetY;  
  let overPort = null;  
  ports.forEach(port => {  
    if (port.isOver(mouseX, mouseY)) {  
      overPort = port;  
      port.isActive = true;  
    }  
  });  
  if (overPort) {  
    overPort.isActive = true;  
  }  
}
```

```
    port.isActive = false;
  }
});
```

```
if (overPort) {
  if (overPort.occupiedShape !== null && overPort.occupiedShape !==
currentlyDragging) {
    overPort.isActive = true;
  }
  else if (overPort.occupiedShape === null) {
    overPort.isActive = true;
  }
}
else
{
  ports.forEach(port => {
    port.isActive = false;
  });
}
drawCanvas();
}
});
```

```
// Attach an event listener to the canvas for the 'mouseup' event
```

```
canvas.addEventListener("mouseup", function(e) {
```

```
  if (currentlyDragging && isDragging) {
```

```
    isDragging = false;
```

```
    const rect = canvas.getBoundingClientRect();
```

```
    const mouseX = e.clientX - rect.left;
```

```
    const mouseY = e.clientY - rect.top;
```

```
    let snappedToPort = false;
```

```
    let previousPort = null;
```

```
    ports.forEach(port => {
```

```
      if (port.occupiedShape === currentlyDragging) {
```

```
        previousPort = port;
```

```
      }
```

```
    });
```

```
    ports.forEach(port => {
```

```
      if (port.isOver(mouseX, mouseY)) {
```

```
        if (port.occupiedShape !== currentlyDragging)
```

```
{  
  if (port.occupiedShape !== null) { // Check if the target port is already occupied  
    currentlyDragging.resetPosition(); // Return dragged shape to original position  
  }  
  else {  
    if (previousPort)  
    {  
      previousPort.occupiedShape = null;  
    }  
    let shapeName = "";  
    let obj = null;  
    currentlyDragging.currX = port.currX;  
    currentlyDragging.currY = port.currY;  
    port.occupiedShape = currentlyDragging;  
    snappedToPort = true;  
    currentlyDragging.inPort = port;  
  
    shapeName = port.getPortShapeID();  
    // console.log(shapeName);  
    obj = stateSwitch(shapeName);  
    console.log(obj);  
  }  
}
```



```
    }  
  }  
  else  
  {  
    snappedToPort = true;  
  }  
}  
else if (port.occupiedShape === null)  
{  
}  
});  
if (!snappedToPort && previousPort)  
{  
  previousPort.occupiedShape = null;  
  currentlyDragging.inPort = null;  
}  
currentlyDragging = null;  
// Reset port colors after drag operation  
ports.forEach(port => port.isActive = false);  
drawCanvas();  
}  
});
```

```
// Attach an event listener to the canvas for the 'mousedown' event
canvas.addEventListener("mousedown", function(e) {

    const rect = canvas.getBoundingClientRect();

    const mouseX = e.clientX - rect.left;

    const mouseY = e.clientY - rect.top;

    let isOverlappingAny = false;

    for (let i = 0; i < rectangles.length; i++) {

        if (isOverlap(mouseX, mouseY, rectangles[i])) {

            // overlapDisplay.innerHTML = "Overlapping";

            isOverlappingAny = true;

            currentlyDragging = rectangles[i];

            isDragging = true;

            dragOffsetX = mouseX - currentlyDragging.currX;

            dragOffsetY = mouseY - currentlyDragging.currY;

            break;

        }

    }

});
```

Result:

```
// get elements from the HTML document using their IDs or selectors

const canvas = document.querySelector("#drag-canvas"); // Selects the canvas
element with the ID "drag-canvas"

const context = canvas.getContext("2d"); // Gets the 2D rendering context of the
canvas

/*

// These lines are commented out, so they are currently not executed.

// They likely intended to select HTML elements to display coordinates and overlap
status.

const coordDisplay = document.querySelector("#coordinates");

const overlapDisplay = document.querySelector("#isOver");

*/

// Variables to manage drag functionality

let currentlyDragging = null; // Stores the currently dragged Rectangle object (null if
nothing is being dragged)

let dragOffsetX = 0; // Stores the horizontal offset between the mouse position and
the dragged object's origin

let dragOffsetY = 0; // Stores the vertical offset between the mouse position and the
dragged object's origin

let isDragging = false; // Boolean flag indicating whether a drag operation is in
progress
```

```

// Class definition for creating the monitor viewport

class Monitor {

    // a monitor needs x, y, width, height, and a highlight color

    constructor (positionX, positionY, width = 100, height = 200, highlightColor =
"orange") {

        this.posX = posX; // X-coordinate of the monitor's top-left corner

        this.posY = posY; // Y-coordinate of the monitor's top-left corner

        this.width = width; // Width of the monitor

        this.height = height; // Height of the monitor

        this.highlight = highlightColor; // Color used for highlighting elements within the
monitor

        this.text = ""; // Text content to be displayed on the monitor

        this.description = ""; // Additional description or information related to the monitor's
state

        this.buffer = 100; // Buffer space below the main monitor area for the button and
label

        this.isHover = false; // Boolean flag indicating if the mouse is currently hovering over
the system check button

    }

    // set text and description for the monitor

    setMonitorValues(text, description) {

        this.text = text;

        this.description = description;

```

```
}
```

```
/* Solution courtesy of  
https://stackoverflow.com/questions/2936112/text-wrap-in-a-canvas-element */  
  
// Helper function to wrap text within a given width on the canvas  
getLines(ctx, text, maxWidth) {  
    let words = text.split(" "); // Split the text into an array of words  
    let lines = []; // Array to store the lines of wrapped text  
    let currentLine = words[0]; // Start with the first word as the initial line  
  
    // Iterate through the remaining words  
    for (let i = 1; i < words.length; i++) {  
        let word = words[i];  
  
        // Measure the width of the current line plus the next word  
        let width = ctx.measureText(currentLine + " " + word).width;  
  
        // If adding the word doesn't exceed the maximum width, add it to the current line  
        if (width < maxWidth) {  
            currentLine += " " + word;  
        } else {  
            // Otherwise, push the current line to the array of lines and start a new line with  
            // the current word  
            lines.push(currentLine);  
            currentLine = word;  
        }  
    }  
    lines.push(currentLine);  
}
```

```

    }
}

// Push the last current line to the array of lines
lines.push(currentLine);

return lines; // Return the array of wrapped text lines
}

// draw the entire monitor, starting with outer board and going to inner board
drawBoard(ctx, text) {

    this.drawBoardOuter(ctx); // Draw the outer tan rectangle of the monitor

    this.drawBoardInner(ctx, "Welcome!", text); // Draw the inner black rectangle and
display text
}

// draw the inner rectangle for the monitor
drawBoardInner(ctx, title = null, text = null, fillColor = "") {

    // set a buffer for the inner rectangle of the monitor
    let buffer = 15;

    let innerX = this.posX + buffer; // X-coordinate of the inner rectangle
    let innerY = this.posY + buffer; // Y-coordinate of the inner rectangle
    let innerWidth = this.width - (2*buffer); // Width of the inner rectangle
    let textX = innerX + 10; // Starting X-coordinate for text
    let textY = innerY + 25; // Starting Y-coordinate for text

```

```
// inner rectangle

if(fillColor) ctx.fillStyle = fillColor; // Set fill color if provided

else ctx.fillStyle = "black"; // Default fill color is black

ctx.strokeRect(innerX, innerY, innerWidth, this.height - (2*buffer)); // Stroke the
inner rectangle

ctx.fillRect(innerX, innerY, innerWidth, this.height - (2*buffer)); // Fill the inner
rectangle

ctx.strokeRect(this.posX, this.posY + this.height, this.width, this.buffer); // Stroke
the buffer area below the monitor

// initialize variables for generating text

let textFont = 20; // Default font size

let defaultText = "Add or remove a component to get started! Once you are done,
you can click the system check button to see if your computer works!"; // Default text
displayed on the monitor

let formattedText = null; // Variable to hold the wrapped text lines

let screenLength = innerWidth; // Available width for text


// set the ctx fill style and font for the title

ctx.fillStyle = "lime"; // Set text color to lime green

ctx.font = `${textFont}px Arial`; // Set default font


// add title if exists

if(title != null) {

    textFont = 36; // Increase font size for the title
```

```
textY += 20; // Adjust Y-coordinate for title

ctx.font = `${textFont}px Arial`; // Set font for the title

ctx.fillText(title, textX, textY, screenLength); // Draw the title

textY += textFont; // Update Y-coordinate for subsequent text
}
```

```
// set the ctx fill style and font for regular text
```

```
textFont = 20; // Reset font size for regular text
```

```
ctx.fillStyle = "lime"; // Set text color to lime green
```

```
ctx.font = `${textFont}px Arial`; // Set font for regular text
```

```
// display the normal text field if exists. Else, display default text.
```

```
if (text != null && text.isArray == false) {
```

```
    formattedText = this.getLines(ctx, text, screenLength); // Wrap the provided text
```

```
} else {
```

```
    formattedText = this.getLines(ctx, defaultText, screenLength); // Wrap the default
text
```

```
}
```

```
// Loop through the formatted text lines and draw them on the canvas
```

```
for ( i = 0; i < formattedText.length; i++ ) {
```

```
    if(formattedText.length == 1) {
```

```
        ctx.fillText(formattedText[i], textX, textY + textFont*i, screenLength); // Draw
```


single-line text

```
    } else {  
        ctx.fillText(formattedText[i], textX, textY + textFont*i, screenLength); // Draw  
multi-line text  
    }  
}  
}
```

// draw the outside of the monitor

```
drawBoardOuter(ctx) {  
    // outer rectangle  
    ctx.strokeStyle = "black"; // Set stroke color to black  
    ctx.fillStyle = "tan"; // Set fill color to tan  
    ctx.fillRect(this.posX, this.posY, this.width, this.height); // Fill the outer rectangle  
    ctx.strokeRect(this.posX, this.posY, this.width, this.height); // Stroke the outer  
rectangle  
}
```

// draw the system check button

```
drawButton(ctx) {  
    let circleX = this.posX + this.width * 2 / 3; // X-coordinate of the button's center  
    let circleY = this.posY + this.height + this.buffer/2; // Y-coordinate of the button's  
center  
    let radius = (this.buffer / 2) - 10; // Radius of the button
```

```
ctx.fillStyle = this.highlight; // Set fill color to the highlight color  
ctx.beginPath(); // Start a new drawing path  
ctx.arc(circleX, circleY, radius, 0, 2 * Math.PI); // Draw a circle  
ctx.fill(); // Fill the circle  
ctx.closePath(); // Close the current path  
ctx.stroke(); // Stroke the circle's outline  
}
```

```
// draw the highlights for the system check button when hovered
```

```
drawHighlight(ctx) {
```

```
  if(this.isHover) {
```

```
    // variables to hold starting positions for the highlight lines
```

```
    let startX1 = this.posX + this.width * 2 / 3 - this.buffer / 2;
```

```
    let startY1 = this.posY + this.height + this.buffer / 2;
```

```
    let startX2 = this.posX + this.width * 2 / 3 + this.buffer / 2;
```

```
    let diagY = 5; // Vertical spacing for the highlight lines
```

```
ctx.strokeStyle = this.highlight; // Set stroke color to the highlight color
```

```
// draw the first line (left side)
```

```
ctx.beginPath();
```

```
ctx.fillStyle = "";
```

```
ctx.strokeStyle = this.highlight;

ctx.moveTo(startX1, startY1);

ctx.lineTo(startX1 - 20, startY1);

ctx.closePath();

ctx.stroke();


// draw the second line (left side, below)

ctx.beginPath();

ctx.fillStyle = "";

ctx.strokeStyle = this.highlight;

ctx.moveTo(startX1, startY1 + (4*diagY));

ctx.lineTo(startX1 - 20, startY1 + (6*diagY));

ctx.closePath();

ctx.stroke();


// draw the third line (left side, above)

ctx.beginPath();

ctx.fillStyle = "";

ctx.strokeStyle = this.highlight;

ctx.moveTo(startX1, startY1 - (4*diagY));

ctx.lineTo(startX1 - 20, startY1 - (6*diagY));

ctx.closePath();
```

```
ctx.stroke();
```

```
// draw the first line (right side)
```

```
ctx.beginPath();
```

```
ctx.fillStyle = "";
```

```
ctx.strokeStyle = this.highlight;
```

```
ctx.moveTo(startX2, startY1);
```

```
ctx.lineTo(startX2 + 20, startY1);
```

```
ctx.closePath();
```

```
ctx.stroke();
```

```
// draw the second opposite line (right side, below)
```

```
ctx.beginPath();
```

```
ctx.fillStyle = "";
```

```
ctx.strokeStyle = this.highlight;
```

```
ctx.moveTo(startX2, startY1 + (4*diagY));
```

```
ctx.lineTo(startX2 + 20, startY1 + (6*diagY));
```

```
ctx.closePath();
```

```
ctx.stroke();
```

```
// draw the third opposite line (right side, above)
```

```
ctx.beginPath();
```

```
    ctx.fillColor = "";

    ctx.strokeColor = this.highlight;

    ctx.moveTo(startX2, startY1 - (4*diagY));

    ctx.lineTo(startX2 + 20, startY1 - (6*diagY));

    ctx.closePath();

    ctx.stroke();

}

}

// draw the label for the system check button
drawButtonLabel(ctx) {

    // variables for x and y of text

    let textX = this.posX + this.buffer / 2;

    let textY = this.posY + this.height + this.buffer / 2 + 10;

    // change stroke color and stroke style (although stroke is not used for fillText)

    ctx.strokeColor = "black";

    ctx.strokeStyle = "black";

    ctx.fillStyle = "black";

    // add the text in 30px Arial;

    ctx.font = "30px Arial";

    ctx.fillText("System Check", textX, textY);

}
```

```
// draw an arrow pointing to the system check button

drawArrowToButton(ctx) {

    // variables to determine start of arrow

    let startX = this.posX + this.width / 2 - 20;

    let startY = this.posY + this.height + this.buffer / 2 - 5;

    // set colors to highlight color

    ctx.strokeStyle = this.highlight;

    ctx.strokeStyle = this.highlight;

    ctx.fillStyle = this.highlight;

    // starting position

    ctx.moveTo(startX, startY);

    // start drawing the arrow shape

    ctx.lineTo(startX + 20, startY);

    ctx.lineTo(startX + 20, startY - 10);

    ctx.lineTo(startX + 35, startY + 5);

    ctx.lineTo(startX + 20, startY + 20);

    ctx.lineTo(startX + 20, startY + 10);

    ctx.lineTo(startX, startY + 10);

    ctx.lineTo(startX, startY);

    ctx.closePath(); // Close the path to form a complete shape

    ctx.fill(); // Fill the arrow shape
```

```
// stroke the shape outline
```

```
ctx.stroke();
```

```
}
```

```
// Main draw function for the monitor, calls other drawing methods
```

```
draw(ctx, text) {
```

```
  this.drawButtonLabel(ctx); // Draw the button label
```

```
  this.drawArrowToButton(ctx); // Draw the arrow pointing to the button
```

```
  this.drawButton(ctx); // Draw the system check button
```

```
  this.drawHighlight(ctx); // Draw the hover highlight for the button
```

```
  this.drawBoard(ctx, text); // Draw the monitor board with the provided text
```

```
}
```

```
// Method to check if a given mouse coordinate is inside the bounds of the system  
check button
```

```
isOverButton(mouseX, mouseY) {
```

```
  // get circle position x and y
```

```
  let circleX = this.posX + this.width * 2 / 3;
```

```
  let circleY = this.posY + this.height + this.buffer/2;
```

```
  // get circle radius
```

```
  let radius = (this.buffer / 2) - 10;
```

```
  // get distance x and y using mouse and circle position
```

```
  let distX = mouseX - circleX;
```

```

let distY = mouseY - circleY;

// calculate the hypotenuse via pythagorean theorem
// (h*h) = (a*a) + (b*b) OR h = square root of ((a*a) + (b*b))
let h = Math.sqrt(Math.pow(distX, 2) + Math.pow(distY, 2));

// return whether hypotenuse is smaller than circle radius
return (
    h <= radius
);
}
}

// Class definition for creating rectangle objects (representing computer components)
class Rectangle {
    constructor(width, height, posX, posY, fillColor, text="") {
        this.width = width; // Width of the rectangle
        this.height = height; // Height of the rectangle
        this.currX = posX; // Current X-coordinate of the rectangle's top-left corner
        this.currY = posY; // Current Y-coordinate of the rectangle's top-left corner
        this.fillColor = fillColor; // Fill color of the rectangle
        this.textValue = text; // Text label of the component
        this.originalX = posX; // Store the original X position of the rectangle
        this.originalY = posY; // Store the original Y position of the rectangle
        this.inPort = null; // Keep track of which Port object the rectangle is currently in (null

```



```
if not in a port)
```

```
}
```

```
// Method to draw the rectangle on the canvas
```

```
draw(ctx) {
```

```
    ctx.fillStyle = this.fillColor; // Set the fill color
```

```
    ctx.fillRect(this.currX, this.currY, this.width, this.height); // Draw the filled rectangle
```

```
    if (this.textValue) { // If the rectangle has a text label
```

```
        ctx.fillStyle = "black"; // Set text color to black
```

```
        ctx.font = "12px Arial"; // Set font for the label
```

```
        ctx.fillText(this.textValue, this.currX + this.width/6, this.currY + this.height/2 + 5); //
```

```
Draw the text label centered on the rectangle
```

```
    }
```

```
}
```

```
// Method to reset the position of the rectangle to its original coordinates
```

```
resetPosition() {
```

```
    this.currX = this.originalX;
```

```
    this.currY = this.originalY;
```

```
    this.inPort = null; // Clear the inPort property when resetting
```

```
}
```

```
}
```

```
// Class definition for creating "Port" objects (slots for computer components)

class Port {

    // Removed currStrokeStyle, keep the style consistent

    isActive = false; // Boolean flag indicating if the mouse is currently hovering over the
    port

    occupiedShape = null; // Stores the Rectangle object that is currently occupying this
    port (null if the port is empty)

    constructor(width, height, posX, posY, strokeColor) {

        this.width = width; // Width of the port

        this.height = height; // Height of the port

        this.currX = posX; // X-coordinate of the port's top-left corner

        this.currY = posY; // Y-coordinate of the port's top-left corner

        this.strokeColor = strokeColor; // This is the *fixed* stroke color of the port
    }

    // Method to get the textValue (ID) of the shape currently occupying the port

    getPortShapeID() {

        if (this.occupiedShape != null) return this.occupiedShape.textValue;

        else return null;

    }

    // Method to draw the port on the canvas

    draw(ctx) {
```

```

    ctx.strokeStyle = this.strokeColor; // Use the fixed stroke color

    ctx.strokeRect(this.currX, this.currY, this.width, this.height); // Draw the port's
rectangle

    this.drawIndicator(ctx); // Draw the color indicator next to the port
}

// Method to draw a color indicator next to the port to show its status
drawIndicator(ctx) {

    let indicatorColor = 'red'; // Default color is red (empty or incorrect component)

    if (this.occupiedShape !== null)

    {

        indicatorColor = 'green'; // Green if a component is in the port

    }

    if (this.isActive) {

        indicatorColor = "yellow"; // Yellow if the mouse is hovering over the port

    }

    ctx.fillStyle = indicatorColor; // Set the fill color for the indicator

    ctx.fillRect(this.currX + this.width + 10, this.currY, 10, 10); // Draw a small filled
rectangle as the indicator

}

// Method to check if a given mouse coordinate is inside the bounds of the port
isOver(mouseX, mouseY) {

```

```

return (

    mouseX >= this.currX &&

    mouseX <= this.currX + this.width &&

    mouseY >= this.currY &&

    mouseY <= this.currY + this.height

);

}

}

// Function to check if a given mouse coordinate is inside a given rectangle object
function isOverlap(mouseX, mouseY, rectangle) {

    return (

        mouseX >= rectangle.currX &&

        mouseX <= rectangle.currX + rectangle.width &&

        mouseY >= rectangle.currY &&

        mouseY <= rectangle.currY + rectangle.height

    );

}

////////////////////////////////////

/* Primary driver code block */

////////////////////////////////////

```

```
/*
```

Define constant for potential error/success states

Unless otherwise noted, most responses have been gathered using generative AI feedback */

```
const states = {
```

```
  "cpu": {
```

```
    "success": "We are now capable of reading instructions in our computer.",
```

```
    "error": "The CPU is the brain. We cannot read instructions without it."
```

```
  },
```

```
  /* Details on rom found at https://www.geeksforgeeks.org/read-only-memory-rom/ */
```

```
  "rom": {
```

```
    "success": "This is a recipe book that your computer cannot lose, storing important system instructions within.",
```

```
    "error": "Without rom, the computer will never wake up. It holds important instructions on how to do so."
```

```
  },
```

```
  "ram": {
```

```
    "success": "Ram helps the computer quickly remember items. Adding more can help the computer do jobs faster.",
```

```
    "surplus": "Adding extra ram helps the computer do things faster!",
```

```
    "warning": "Removing ram reduces a computer's ability to think and multitask. It will be slower when using more than one function.",
```

```
    "error": "Without ram, the computer will not work. At least one is needed to help read the instructions on how to boot up."
```

},

"gpu": {

"success": "This is the computer's artist! It helps the computer make images, videos, and games look good.",

"error": "Without the GPU, pictures and videos might look blurry or choppy, and games might not work well."

},

"storage": {

"success": "Putting in the storage drive lets you save all your work and games so you can use them again later.",

"surplus1": "Adding more storage lets the computer remember more. Memory was expensive for computers for a while, but got cheaper very fast.",

"surplus2": "Now the computer can remember a lot. This is probably how much current computers can remember.",

"warning": "Removing memory can cause the computer to remember less. The stuff on the removed component could still be seen.",

"error": "Without the storage drive, you can't save anything, and the computer won't remember anything when you turn it off."

},

"sound": {

"success": "Putting in the sound card lets you hear all the sounds the computer makes.",

"error": "Without the sound card, the computer will be silent – you won't hear any music or sounds."

},

"network": {

"success": "Putting in the network card lets you connect to the internet and talk to other computers.",

"error": "Without the network card, you can't go online or play games with friends on the internet."

},

"system": {

"success": "Success! Your computer is fully functional.",

"surplus": "Success! Not only does it work, your computer is better than before!",

"complete": "Wow! Your computer has all the best components available!",

"error": "Oops! Something is missing from the computer that is needed to function. Try adding more components."

},

}

// Define monitor for the canvas

const monitor = new Monitor((canvas.width / 2) + 20, 60, (canvas.width / 2) - 40, 400, "Red");

// Define the initial properties for the rectangles (draggable components)

const shapes = [[80, 20, 100, 80, 3, "RAM"], // [width, height, startX, startY, count, text]

[80, 20, 40, 200, 1, "GPU"],

[80, 20, 200, 80, 3, "Storage"],

[80, 20, 40, 240, 1, "Network"],

[80, 20, 40, 280, 1, "Sound"],

```

        [40, 40, 40, 80, 1, "CPU"],

        [40, 40, 40, 140, 1, "ROM"]];

// Define the initial properties for the ports (component slots)

const portShapes = [[80, 20, 340, 60, 3], // [width, height, startX, startY, count]

        [80, 20, 480, 60, 3],

        [80, 20, 340, 180, 3],

        [40, 40, 480, 180, 1],

        [40, 40, 480, 240, 1]];

// Define an array of colors to be used for components

const colors = ["pink", "lightblue", "lightgreen", "orange", "cyan", "yellow",
"AntiqueWhite", "Beige"];


// define a variable to hold all known component types and their current count and
limit

let componentCount = {

    "ram": {

        "count": 0,

        "limit": 3,

    },

    "storage": {

        "count": 0,

        "limit": 3,

    },

```



```
"gpu": {  
  "count": 0,  
  "limit": 1,  
},  
"network": {  
  "count": 0,  
  "limit": 1,  
},  
"sound": {  
  "count": 0,  
  "limit": 1,  
},  
"cpu": {  
  "count": 0,  
  "limit": 1,  
},  
"rom": {  
  "count": 0,  
  "limit": 1,  
},  
}
```

```

// define some preliminary variables

let rectangles = []; // Array to hold the Rectangle objects (draggable components)

let ports = []; // Array to hold the Port objects (component slots)

let i = 0; // Loop counter


// Loop through the 'shapes' array to create Rectangle objects and add them to the
'rectangles' array

for (i = 0; i < shapes.length; i++) {

    for (let j = 0; j < shapes[i][4]; j++) { // Create multiple rectangles based on the count
in 'shapes'

        let rect = new Rectangle(shapes[i][0], shapes[i][1], shapes[i][2], shapes[i][3] + (j *
40), colors[i % colors.length], shapes[i][5]);

        rectangles.push(rect);

    }

}


// Loop through the 'portShapes' array to create Port objects and add them to the
'ports' array

for (i = 0; i < portShapes.length; i++) {

    for (let j = 0; j < portShapes[i][4]; j++) { // Create multiple ports based on the count in
'portShapes'

        let port = new Port(portShapes[i][0], portShapes[i][1], portShapes[i][2],
portShapes[i][3] + (j*40), "black");

        ports.push(port);

        // console.log(` ${i}, ${j}: (i, j)`);

```

```
}  
}
```

// function to alter component count based on object and value (currently not fully implemented or used)

```
function componentModifier(obj, name, mod="") {  
  let comp = name.toLowerCase();  
  if (mod == "" || obj == null || name == "") return;  
  if (mod == "+" && (obj.comp.count + 1 <= obj.comp.limit)) {  
    obj.comp.count += 1;  
    return "pos";  
  } else if (mod == "-" && (obj.comp.count + 1 <= obj.comp.limit)) {  
    obj.comp.count -= 1;  
    return "neg";  
  } else {  
    return null;  
  }  
}
```

// function to get the state object from the 'states' constant based on the component name

```
function stateSwitch(state) {  
  // init variable to hold state object
```

```
let actualState = null;

let str = state.toLowerCase();

/* get the appropriate state based on the given entry value */
if(str == "ram") { // if ram get ram state
    actualState = states.ram;
} else if (str == "storage") { // if storage get storage state
    actualState = states.storage;
} else if (str == "cpu") { // if cpu get cpu state, etc.
    actualState = states.cpu;
} else if (str == "rom") {
    actualState = states.rom;
} else if (str == "gpu") {
    actualState = states.gpu;
} else if (str == "network") {
    actualState = states.network;
} else if (str == "sound") {
    actualState = states.sound;
} else if (str == "system") {
    actualState = states.system;
}
```

```
// return the actualState; if no object, it returns null  
  
return actualState;  
  
}
```

```
// function to draw the monitor with the given text input  
  
function drawMonitor(textInput) {  
  
    if(monitor != null) {  
  
        monitor.draw(context, textInput);  
  
    }  
  
}
```

```
// function to draw the labels for the component, port, and monitor sections  
  
function drawDividerLabels(ctx, width, height) {  
  
    // variables for x and y of text  
  
    let componentX = width * (1/16);  
  
    let portX = width * (11/32);  
  
    let monitorX = width * (21/32);  
  
    let distY = 40;  
  
    // change stroke color and stroke style (although stroke is not used for fillText)  
  
    ctx.strokeStyle = "black";  
  
    ctx.strokeStyle = "black";  
  
    ctx.fillStyle = "black";
```

```
// add the text in 30px Arial;

ctx.font = "30px Arial";

ctx.fillText("Components", componentX, distY);

ctx.fillText("Ports", portX, distY);

ctx.fillText("System Monitor", monitorX, distY);

}
```

// function to draw the vertical lines that divide the canvas into sections

```
function drawDividers() {

    // create variables to hold canvas width, height, and buffer value

    let w = canvas.width;

    let h = canvas.height;

    let buffer = 10;

    // set context styles for the background

    context.fillStyle = "lightgrey";

    context.strokeStyle = "lightgrey";

    context.fillRect(0, 0, w, h);

    // reset context styles for the dividers

    context.fillStyle = "tan";

    context.strokeStyle = "tan";

    context.lineWidth = 10;

    // draw a rectangle around the entire canvas
```

```

context.strokeRect(0, 0, w, h);

// draw a dividing line between the components area and the ports area
context.fillRect(w/4, buffer, 10, h - (buffer * 2));

// draw a dividing line between the ports area and the monitor area
context.fillRect(w/2, buffer, 10, h - (buffer * 2));

// reset line width to default
context.lineWidth = 1;

drawDividerLabels(context, w, h); // Draw the labels for each section
}

// Function to iterate through and draw all the port and rectangle objects
function drawRectangles() {
  ports.forEach(port => {
    port.draw(context);
  });
  rectangles.forEach(rectangle => {
    rectangle.draw(context);
  });
}

// Main function to clear the canvas, draw the dividers, the monitor, and all the
components and ports

```

```

function drawCanvas(textInput) {

  context.clearRect(0, 0, canvas.width, canvas.height); // Clear the entire canvas

  drawDividers(); // Draw the dividing lines

  drawMonitor(textInput); // Draw the system monitor

  drawRectangles(); // Draw all the components and ports

}

// Initial call to the drawCanvas function to set up the initial state of the display
drawCanvas("");

// Function to update the coordinates displayed (currently commented out in the
// mousemove listener)

function updateCoordinates(event) {

  const rect = canvas.getBoundingClientRect();

  const x = event.clientX - rect.left;

  const y = event.clientY - rect.top;

  // coordDisplay.textContent = `X: ${Math.floor(x)}, Y: ${Math.floor(y)}`;

}

// Attach an event listener to the canvas for the 'mousemove' event to handle hover
// effects and dragging

canvas.addEventListener("mousemove", (e) => {

  // update coordinates

  // updateCoordinates(e);

```



```
const rect = canvas.getBoundingClientRect();

const mouseX = e.clientX - rect.left;

const mouseY = e.clientY - rect.top;


// Check if the mouse is over the system check button on the monitor
if (monitor.isOverButton(mouseX, mouseY)) {

    monitor.isHover = true; // Set hover flag to true
} else {

    monitor.isHover = false; // Set hover flag to false
}


// If not currently dragging, check for hover over ports to highlight them
if (!isDragging) {

    ports.forEach(port => {

        if (port.isOver(mouseX, mouseY)) {

            port.isActive = true; // Set active flag for the hovered port
        } else {

            port.isActive = false; // Reset active flag for other ports
        }

    });

    drawCanvas(); // Redraw the canvas to show hover highlights on ports and the
    monitor button
```

```
}
```

```
// If a component is being dragged, update its position
```

```
if (currentlyDragging && isDragging) {
```

```
    currentlyDragging.currX = mouseX - dragOffsetX; // Update the dragged  
    component's X-coordinate
```

```
    currentlyDragging.currY = mouseY - dragOffsetY; // Update the dragged  
    component's Y-coordinate
```

```
    let overPort = null; // Variable to store the port the dragged component is currently  
    over
```

```
    ports.forEach(port => {
```

```
        if (port.isOver(mouseX, mouseY)) {
```

```
            overPort = port; // Assign the hovered port to overPort
```

```
            port.isActive = true; // Highlight the port being hovered over
```

```
        }
```

```
    else{
```

```
        port.isActive = false; // Ensure other ports are not highlighted during drag
```

```
    }
```

```
});
```

```
// If the dragged component is over a port
```

```
if (overPort) {
```

```
    // If the port is occupied by a different component, keep it highlighted
```

```
    if (overPort.occupiedShape !== null && overPort.occupiedShape !==
```

```

currentlyDragging) {
    overPort.isActive = true;
}

// If the port is empty, highlight it
else if (overPort.occupiedShape === null) {
    overPort.isActive = true;
}
}

// If the dragged component is not over any port, unhighlight all ports
else
{
    ports.forEach(port => {
        port.isActive = false;
    });
}

drawCanvas(); // Redraw the canvas to show the dragged component and port
highlights
}

});

// Attach an event listener to the canvas for the 'mouseup' event to handle dropping of
components

canvas.addEventListener("mouseup", function(e) {

```

```
if (currentlyDragging && isDragging) {
```

```
  isDragging = false; // Reset the dragging flag
```

```
  const rect = canvas.getBoundingClientRect();
```

```
  const mouseX = e.clientX - rect.left;
```

```
  const mouseY = e.clientY - rect.top;
```

```
  let snappedToPort = false; // Flag to indicate if the component was successfully  
  dropped into a port
```

```
  let previousPort = null; // Variable to store the port the component was in before the  
  drag
```

```
  // Find the port the currently dragged shape was in before the drag started
```

```
  ports.forEach(port => {
```

```
    if (port.occupiedShape === currentlyDragging) {
```

```
      previousPort = port;
```

```
    }
```

```
  });
```

```
  // Iterate through all the ports to check if the dragged component is dropped over  
  one
```

```
  ports.forEach(port => {
```

```
    if (port.isOver(mouseX, mouseY)) {
```

```
      // If the mouse is over a port
```

```
      if (port.occupiedShape !== currentlyDragging)
```

```

{
    // If the port is already occupied by a different component
    if (port.occupiedShape !== null) {
        currentlyDragging.resetPosition(); // Return the dragged component to its
original position
    }

    // If the port is empty
    else {
        // If the component was in a port before, clear that port
        if (previousPort)
        {
            previousPort.occupiedShape = null;
        }

        let shapeName = "";

        let obj = null;

        currentlyDragging.currX = port.currX; // Snap the dragged component's X to
the port's X

        currentlyDragging.currY = port.currY; // Snap the dragged component's Y to the
port's Y

        port.occupiedShape = currentlyDragging; // Set the port's occupied shape to
the dragged component

        snappedToPort = true; // Set the snapped flag to true

        currentlyDragging.inPort = port; // Update the component's inPort property

```

```
    shapeName = port.getPortShapeID(); // Get the ID (textValue) of the
component

    // console.log(shapeName);

    obj = stateSwitch(shapeName); // Get the state object for the component type
    console.log(obj);


    // Update the component count
    if (obj) {

        const componentType = shapeName.toLowerCase();

        if (componentCount.hasOwnProperty(componentType)) {

            componentCount[componentType].count++;

        }

    }

}

// If the dragged component was already in this port, consider it snapped
else

{

    snappedToPort = true;

}

}

// If the mouse is not over this port and the port was previously occupied by the
dragged shape, clear the hover state
```

```
else if (port.occupiedShape === currentlyDragging)
{
    port.isActive = false;
}
});
```

// If the component was not snapped to any port and it was in a port before, clear its inPort status

```
if (!snappedToPort && previousPort)
{
    previousPort.occupiedShape = null;
    currentlyDragging.inPort = null;

    // Update the component count if removed from a port
    const componentType = currentlyDragging.textValue.toLowerCase();
    if (componentCount.hasOwnProperty(componentType) &&
componentCount[componentType].count > 0) {
        componentCount[componentType].count--;
    }
}
```

currentlyDragging = null; // Reset the currently dragging variable

// Reset port colors after drag operation

```

    ports.forEach(port => port.isActive = false);

    drawCanvas(getMonitorText()); // Redraw the canvas, updating the monitor text
}

});

// Attach an event listener to the canvas for the 'mouseup' event
canvas.addEventListener("mouseup", function(e) {

    // Check if a component was being dragged when the mouse button was released
    if (currentlyDragging && isDragging) {

        isDragging = false; // Reset the dragging flag

        const rect = canvas.getBoundingClientRect();

        const mouseX = e.clientX - rect.left; // Get the x-coordinate of the mouse click
        relative to the canvas

        const mouseY = e.clientY - rect.top; // Get the y-coordinate of the mouse click
        relative to the canvas

        let snappedToPort = false; // Flag to track if the dragged component was
        successfully dropped into a port

        let previousPort = null; // Variable to store the port the component was in before the
        drag started

        // Iterate through all the ports to find if the currently dragged shape was occupying
        one

        ports.forEach(port => {

```



```
if (port.occupiedShape === currentlyDragging) {  
    previousPort = port; // Store the port the component was previously in  
}  
});
```

```
// Iterate through all the ports to check if the mouseup event occurred over one  
ports.forEach(port => {  
    if (port.isOver(mouseX, mouseY)) {  
        // If the mouse is over a port  
        if (port.occupiedShape !== currentlyDragging)  
        {  
            // If the target port is already occupied by a different component  
            if (port.occupiedShape !== null) {  
                currentlyDragging.resetPosition(); // Return the dragged component to its  
original starting position  
            }  
            // If the target port is empty  
            else {  
                // If the dragged component was in a port before, clear that port's occupied  
status  
                if (previousPort)  
                {  
                    previousPort.occupiedShape = null;
```

```

    }

    let shapeName = ""; // Variable to store the name of the component

    let obj = null; // Variable to potentially store a related object (not fully utilized
in this snippet)

    currentlyDragging.currX = port.currX; // Set the dragged component's
x-coordinate to the port's x-coordinate, effectively snapping it

    currentlyDragging.currY = port.currY; // Set the dragged component's
y-coordinate to the port's y-coordinate, effectively snapping it

    port.occupiedShape = currentlyDragging; // Set the port's 'occupiedShape'
property to the currently dragged component

    snappedToPort = true; // Set the flag to indicate that the component was
snapped to a port

    currentlyDragging.inPort = port; // Update the 'inPort' property of the dragged
component to reference the port it's now in


    shapeName = port.getPortShapeID(); // Get the identifier (textValue) of the
component that was just dropped

    // console.log(shapeName);

    obj = stateSwitch(shapeName); // Call the function to get the state object
based on the component name

    console.log(obj); // Log the state object (for debugging or potential future use)

    }

    }

    // If the mouseup occurred over the same port the component was already in,
consider it snapped

```

```

else
{
    snappedToPort = true;
}
}

// If the mouse is not over this port and this port was previously occupied by the
currently dragged shape

else if (port.occupiedShape === currentlyDragging)
{
    port.isActive = false; // Deactivate the port's hover/active state
}

});

// If the component was not snapped to any port after dragging and it was in a port
before the drag

if (!snappedToPort && previousPort)
{
    previousPort.occupiedShape = null; // Clear the previous port's occupied status

    currentlyDragging.inPort = null; // Clear the 'inPort' property of the dragged
component
}

currentlyDragging = null; // Reset the 'currentlyDragging' variable as the drag
operation has ended

```

```

// Reset the 'isActive' property of all ports to false after the drag operation
ports.forEach(port => port.isActive = false);

drawCanvas(); // Redraw the canvas to reflect the changes (component positions,
port states)
}
});

// Attach an event listener to the canvas for the 'mousedown' event
canvas.addEventListener("mousedown", function(e) {

  const rect = canvas.getBoundingClientRect();

  const mouseX = e.clientX - rect.left; // Get the x-coordinate of the mouse click
  relative to the canvas

  const mouseY = e.clientY - rect.top; // Get the y-coordinate of the mouse click
  relative to the canvas

  let isOverlappingAny = false; // Flag to check if the mouse click overlapped with any
  draggable rectangle

  // Iterate through the array of rectangles (draggable components)
  for (let i = 0; i < rectangles.length; i++) {

    // Check if the mouse click coordinates are within the bounds of the current
    rectangle

    if (isOverlap(mouseX, mouseY, rectangles[i])) {

      // overlapDisplay.innerHTML = "Overlapping"; // Uncomment this line if you have an
      element with this ID to display overlap status

      isOverlappingAny = true; // Set the flag to true as an overlap occurred

```

```
    currentlyDragging = rectangles[i]; // Set the 'currentlyDragging' variable to the  
    rectangle that was clicked
```

```
    isDragging = true; // Set the 'isDragging' flag to true to indicate a drag operation  
    has started
```

```
    dragOffsetX = mouseX - currentlyDragging.currX; // Calculate the horizontal offset  
    between the mouse click and the rectangle's top-left corner
```

```
    dragOffsetY = mouseY - currentlyDragging.currY; // Calculate the vertical offset  
    between the mouse click and the rectangle's top-left corner
```

```
    break; // Exit the loop as we have found the rectangle that was clicked and started  
    dragging
```

```
    }
```

```
    }
```

```
});
```