Distributed Displays for Discrete Integrated Circuit Electronics

by

Justin Browning Christensen

B.S., Brigham Young University (2019)

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning, in partial fulfillment of the requirements for the degree of

Master of Science in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2021

Author .....
Program in Media Arts and Sciences
August 20, 2021

Certified by .....
Neil Gershenfeld
Director, The Center for Bits and Atoms
Thesis Supervisor

Accepted by .....
Tod Machover
Academic Head, Program in Media Arts and Sciences

Distributed Displays for Discrete Integrated Circuit Electronics

by

Justin Browning Christensen

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning, on August 20, 2021 in partial fulfillment of the requirements for the degree of

Master of Science in Media Arts and Sciences

Abstract

I present a distributed display architecture that integrates with a set of asynchronous mechanically and electrically re-configurable computing nodes, otherwise known as Discrete Integrated Circuit Electronics (DICE). Each singular display is physically and electrically connected to a DICE node which transmits useful data to be displayed. By integrating these displays with the DICE nodes, a multitude of applications are enabled, starting with real-time data visualization and debugging, and going on up to more complex applications, such as locally computed ray tracing and graphics rendering, as well as structural and volumetric displays. The advantages and implementation of the displays into the DICE architecture, as well as various examples of their applications are demonstrated and discussed. While the DICE nodes themselves address issues with locality in computing, these integrated distributed displays will help them overcome some of their limitations and enhance their capabilities. Together, these integrated devices and their scalability can lead to iterative improvements to graphical processing, form into spatial 2D grid (structural) and 3D mesh (volumetric) displays, and overall reduce the cost and complexity of distributed display and computing systems.

Thesis Supervisor: Neil Gershenfeld
Title: Director, The Center for Bits and Atoms

Distributed Displays for Discrete Integrated Circuit Electronics

by

Justin Browning Christensen

This thesis has been reviewed and approved by the following committee members:

Neil Gershenfeld .....
Director, The Center for Bits and Atoms
Professor, Media Arts and Sciences
Massachusetts Institute of Technology

Joseph Paradiso .....
Professor, Media Arts and Sciences
Massachusetts Institute of Technology

Cardinal Warde .....
Professor, Electrical Engineering
Massachusetts Institute of Technology

Acknowledgments

I would not be where I am today without the tremendous love and support provided to me by so many people. I hope to express even a fraction of the gratitude I feel to those that made all of this possible.

First off, thank you to my advisor, Neil, for all that you've done for me. From opening up your group to me when I needed to find a new home in the lab, to all of the advice and counsel you've given to me. Beyond the technical portions that you helped me with, the habits and life skills you taught me are sure to be of great value in my life. Thank you for providing such a wonderful place to learn and grow in the Center for Bits and Atoms. I've had access to tools and skills I never would have before, and my mind has been expanded greatly because of it.

Thank you to my readers, Joe Paradiso and Cardinal Warde. Your suggestions and guidance over the past months have been invaluable in shaping this work and keeping me on track. Joe, thank you for being such an inspiring figure within the Media Lab with all the work you do. What stands out to me the most, however, is your willingness to meet with any student, no matter how busy your schedule is. Cardinal, thank you for all of the chats we've had about holography and display technology, and for being such a great instructor. I'm very glad I took your class when I first arrived at MIT.

Thank you to the rest of CBA for inspiring me to be better. Kara, Sherry, Joe, Candace, and James, you do a tremendous job keeping the lab and its subsequent functions running. Tom and John, thank you so much for being incredible teachers and friends in the shop. I loved spending my time in there with you learning to make new things and fix the machines that make them. Pranam, Ben, Jake, Amira, Filippos, Patricia, Jiri, Chris, Alfonso, Camron, and David, thank you for being so helpful and welcoming me into the group. I've really enjoyed seeing the work you do and being a part of it.

Thank you in particular to Zach and Erik. You both have been absolute life-savers to me. Many sleepless nights potentially banging my head against the wall (perhaps both figuratively and literally) were avoided due to your generosity and helpfulness with this project. Zach, you taught me so many tricks and tips to hone in my hardware development skills. Erik, you helped me understand the code and programming as a whole so much better. You two deserve a lot of credit for empowering me to clear the obstacles I was faced with. You were so responsive and helpful at any time of day, and even while on vacation. For that, I express my utmost appreciation. Thank you for being my friends and being so kind and generous to me with your time.

Thank you to my Object-Based Media family. As my original home in the lab, you helped me to reach for the stars and develop that sense of wonder that is so prevalent at the Media Lab. Dan, Kristin, and Catherine, thank you for running the ship and making sure all of us were taken care of. Thank you for all our chats that helped expand my mind and fall in love with the magic of the lab. Pip, Everett, Laura, Bianca, Pedro, and Ali, thank you for being my big brothers and sisters at the lab, helping me get settled in, and being a great place of support for me as I learned

the ropes. Nina, Tyler, and Aubrey, we went through a lot together, and I couldn't ask for a better Masters Squad to be a part of. Thank you so much for everything, you will forever be my dear friends.

Thank you to all the other friends I made at MIT and to the Cambridge 1st Ward. Jonny and Annie, it was so fun for our kids to play together, and I am so grateful for the deep friendships that were formed between our families. I hope it will continue for a long time. Also, thanks for letting us borrow your car so often... Zach and Hailey, thank you for the rides to church, the fun times hanging out, and for always being willing to lend a helping hand. Ryan and Sarah, thanks for being such great neighbors and helping us get settled into Westgate, we're so glad to have met and gotten to know you.

Thank you, Dr. Daniel Smalley, for everything you've done for me. You pioneered the path for this kid from Utah to make it all the way to the MIT Media Lab. From when I first started working with your research group, I had countless memories that I'll cherish forever. I loved making my first hologram, my first holo-chip, being part of publishing a paper for the first time, and so much more. Most of all, though, I'll always remember that day in your office when we first met, which was the moment that I knew that I wanted to become an electrical engineer. You've changed my life for the absolute best, and I'm eternally grateful for it.

Thank you to the absolute best friends in the entire world. Cameron, Amy, Colby, Elias, and Inaya, you have been the greatest support group, cheerleading team, and shoulders to stand on. I still marvel to this day how lucky I am to have you in my life. Whether back in Utah or right there with me in Massachusetts, you all helped motivate me to be my best, and I wouldn't be anywhere close to the person I am today without you. I'm looking forward the rest of our lives and the friendship that will surely make it all so great.

Finally, thank you so much to my parents, grandparents, siblings, in-laws, and the rest of my family. Mom and Dad, you showed me how to work hard and do the best job possible from the very beginning. I strive each day to be the best person I can be and to make you proud, and I hope I've done so. Eric and Terri, thank you for being so supportive, even as I took your daughter and moved her all the way across the country with me. You've been such a blessing in my life, and I can't thank you enough for all you've done. Lia and Millie, Daddy loves you so much, and I'm so grateful for all the laughs, smiles, and play time that we've shared together. You brighten my day always and give me so many reasons to keep pushing and working. Kayla, you are the love of my life and I'd be completely lost without you. We've navigated so many decisions and trials in life together already, and there's no one else I'd rather experience it all with. Thank you for loving me and inspiring me to be better, thank you for putting up with and helping me overcome my faults, and thank you for helping me achieve this great accomplishment. I don't know where the rest of life will take us, but it doesn't matter so much as long as I get to hold your hand to walk beside you down that path as we let God prevail in our lives.

Thank you, to each and every one of you, and to any others that have been a part of my journey. I wouldn't be here without you, and I hope to continue on to make you all proud and to make this world a better place.

Contents

Chapter 1

Introduction

The Discrete Integrated Circuit Electronics (DICE) project is an effort by the MIT Center for Bits and Atoms to create an alternative computing architecture that is comprised of a series of interconnected "nodes" that are discretely assembled to perform a wide variety of parallel computational tasks [12]. Rather than the traditional definition of the point where two wires connect together, the word "node" here means a singular, self-contained device capable of performing computations, but with the capability of communicating its results and other data to other nodes. The work of this thesis is to take this new architecture and enhance it by integrating a set of distributed displays, enabling real-time debugging and data visualization, along with the plethora of applications opened up by these additions. The merits of these integrated displays and the issues that they overcome will be discussed, but first, the motivation and history behind the DICE project will be explained.

1.1 Background

Modern computational architectures have various limitations, such as minimum feature size, tunneling effects, and the ongoing pursuit of mastery over quantum effects. Advances in the industry have been able to avoid some of these limitations by focusing on refining other aspects and features. For example, with the focus on shrinking integrated circuit components according to Moore's Law, the complexity of comput-

ing devices has increased while the size or cost have either decreased or stayed the same. However, as technology approaches the limits of what is currently possible, it's not exactly clear which direction new advances can even take.

One of these limitations, which has been largely avoided by focusing on shrinking component size, comes in the form of how fast data can travel. Light can travel approximately one foot in a single nanosecond. Considering that up to gigahertz clock rates are often used in modern processors, that doesn't give a signal much time to access the appropriate memory cache and complete its assigned tasks. As the size of a circuit increases, the likelihood of being able to access a certain cell of memory anywhere on the device within a given clock cycle deteriorates. This can especially be seen in large super-computing centers. Because of the physical distance between different servers, quite complex routing and networking is required to ensure that each unit has access to the appropriate memory lines.

The above example, along with the rest of current common architectures, highlights a flaw that has largely gone under the radar: dependence on being able to access the same global memory from anywhere within a system. Because of developments according to Moore's Law, the distance between components within a circuit have decreased fairly proportionally to the increase in the complexity of the systems. Therefore, that physical limit of two components being too far apart to communicate within the clock cycle has been skirted. Thinking more in depth about this, though, causes one to question the necessity of this dependence on a global memory, which is present even among most modern architectures. While we haven't had to worry about running into this limitation, achieving even smaller feature sizes is becoming increasingly difficult and perhaps there isn't much more room to grow (or rather, shrink) down this current path. [25]

1.2 Locality

Over the years, various scholars have called into question this dependence on a global memory line and have sought to come up with architectures that bypass the need for

such universal access. Imagine if instead of one large system with different specialized areas that all needed to be in communication with one another, there were multiple instances of the same computing structure that were self-contained but could communicate with its closest neighbor. This method brings up the idea of "locality" in computing. If a single unit is able to handle computations at a local level, then share its results only with units that need them, not only does it remove the need for a global memory shared between every single unit, but it also means that the system can be scaled to meet the needs of the application.

One very notable example of this is the CAM-8 computer architecture [20] introduced in the early 1990s. This system's architecture is based on cellular automata, which essentially means that the system is made up of multiple smaller structures that tessellate together both physically and computationally to form one larger, more complex, system. Such a model was particularly useful for a handful of specific applications, most notably of gas particle simulations. Each physical "cell" is responsible for computations within the same area that it represents spatially. Just as gas particles interact with each other locally, each cell simulates the interactions locally, then shares information only with neighboring cells to which the effects of the particles "cross over" to effect them. As each cell runs its local computations, they all come together to form one larger environment that is capable of many feats that aren't as possible or are less efficient when using traditional computing architectures.

An iteration that evolved from this concept was that of a "paintable" computer, [6] which is essentially the idea of filling a sort of paint-like substance with hundreds of tiny and independent microprocessors. These "computing particles" are randomly distributed throughout the paint and are capable of asynchronous computations and local communication. This mixture can then be "painted" onto a substrate, forming the computer's structure according to the topology of the surface. The computing particles can distribute computational tasks to process in parallel, then report their results to their nearest neighbors, summing into a finalized calculation or completed overall computational task.

In a similar sense, there have been a variety of implementations of systems where

multiple computational or sensing devices are connected together to make computational substrates. One example of this is the Chainmail project, where a series of circuit boards with embedded processors and arrays of sensors are all connected together using conductive, but flexible materials to create a flexible skin. [21] A more recent example of this idea that evolved into full on textiles can be found in the Electronic Textile *Gaia* project, which draws upon the rich history of computational substrates to create a system that implements smart textiles at large breadth of scales. [26] Both of these examples implement structures that work independently in parallel, then share the data throughout for computational or sensing tasks.

While these are all very specific implementations of this concept, there is a large history of other work in this field, sometimes referred to as processor-in-memory. These systems work to maximize the efficiency of memory bandwidth by combining the processor and memory instead of keeping them separate like in traditional computational architectures. There have been various successful implementations of these systems, such as Computational RAM, which competes in cost and performance with traditional DRAM, [9] as well as the Terasys prototype, which was capable of supercomputer performance at a fraction of the cost. [13] One more successful implementation is Gilgamesh, which uses a "massively parallel architecture" to achieve up to petaflops of computing performance. [23]

Finally, like all these examples, the Discrete Integrated Circuit Electronics (DICE) project [12], which this work is based upon, completely challenges the idea of having one global clock cycle orchestrating all of the computational tasks. This comes in the form of a set of asynchronous, mechanically and electrically re-configurable computing nodes that are only capable of sending data to and receiving data from their immediate neighbors. By embracing locality in the computations at its foundation, this architecture handles each task by breaking it up into smaller and localized chunks, then bringing it all together for a final result. This system can easily be scaled simply by adding more nodes. In their current form, however, their are some limitations to their effectiveness in developing intriguing applications that need to be addressed. This work focuses on overcoming one of those major obstacles: the need

for visualization within the applications.

1.3 Need for Visualization

While much work has been put into validating and optimizing these nodes and how they communicate, there is still a big drawback within the current implementation of DICE: the lack of visualization of the performance/computations of each node. It is possible to view the data post-computation by porting it out through added signal lines in the system, but it isn't possible to see it in real time. This makes troubleshooting the system difficult, as well as makes it near impossible to visualize any meaningful data until after the simulation is complete, which misses out on perhaps the most interesting and useful portions of certain applications.

The work presented in this thesis is integrating into the system a series of distributed displays at an individual node level, enabling the real time visualization of node data. Using commercial off the shelf displays and connecting them to the already established system, the data is sent to and shown on the displays, enabling real time visualization and debugging of the nodes themselves. On top of these simple functions, these distributed displays provide even more potential applications, a couple of which are explored in depth in this thesis, such as particle simulations and locally computed ray tracing in graphics rendering. The performance of this system versus other commercially available graphics-focused processors is then discussed, showing both the benefits and drawbacks of this system.

The DICE architecture already helps in solving the issue presented by modern computing architectures in being overdependent on having access to a global memory, but by adding these displays to the system, a wide variety of applications are enabled to be explored in the future. With all of the many local computations and visualizations happening in parallel, the nodes can effectively share the load and fundamentally change the way we approach computing. This thesis shows the first steps in that direction, providing concrete examples, as well as theoretical applications to be implemented, explored, and evaluated in the future.

1.4 Distributed Displays

As the main tangible addition to the project that this work brings about, it is necessary to mention current advances in distributed display technology and applications. Distributed displays, in their most basic state, are "computer systems that present output to more than one physical display." [17] Most often, they are seen as large, tiled displays in public spaces or even in TV studios, usually breaking down one source into a separate stream for each display, which combine together to make one large displayed version of the original source. This is typically done by using some sort of software to take the signal, cut it up visually and map it out to the various displays, then send all that data to some hardware that then multiplexes the signals out to each of the appropriate displays. Some use cases that have been further explored is using such displays to be used in offices and conference rooms or similar types of environments. [16] These use cases, as well as others, have typically been approached by researchers implementing their own distributed graphics systems using various computers and other devices networked together to render their displays, even making their process and algorithms scalable for larger displays. [16, 15]

Many of these systems still run into issues, though, especially when dealing with extremely large data sets to process through to render the displays. Often times, they must utilize high-performance computing (HPC) centers to handle the complex data, which takes time and doesn't allow for easily implemented real-time systems. Some have used real-time graphics accelerators, such as those found in supercomputers and gaming consoles, to try to overcome this problem. [10, 19] While a viable solution, the dependence on various external hardware can get messy, and interfacing with it all can be an even bigger problem.

While many modern implementations of distributed displays go big, the implementation with the DICE system will be at a much smaller form factor to match the size of the DICE nodes they will be integrating with. The reason for this is so that the software applications implemented within the hardware can align, especially in applications where the physical node represents the spatial computational volume it

is representing. Many of the same traits found between this system and others are still present, however, such as ability to be tiled and scaled, and general approach to distributing the computation and rendering among the different displays. By utilizing the DICE architecture for computation, the efficiency of the parallel processing should simplify and help overcome the limitations faced by typical distributed display systems.

1.5 Contributions

The contributions that this work has added are as follows: 1) a simple hardware/firmware integration process between the DICE nodes and the displays, 2) a basic framework for visualizing data and debugging information from the nodes to the displays, 3) demonstrable examples of various visualized applications, and 4) a proposed method of computing ray tracing algorithms locally within each node and displaying the subsequent results, as well as performance projections for how this method compares to commercially available components at its current state and in future scaled up versions of this project. While there is still much to be explored on the topic, hopefully these contributions will help continue down the path to discovering more efficient display and computational systems.

Chapter 2

Hardware Integration

2.1 DICE Design

Before diving into how the displays are integrated into the system, a description of the existing DICE hardware is necessary. DICE has evolved as a project, having various iterations depending on the specific application/size constraints. Static-DICE is a set of nodes that are connected in four directions, all on a single PCB. Tiny-DICE is a set of modular nodes made using the smallest commercially available connectors that tessellate in a tetrahedral configuration. Finally, Meso-DICE, which this work builds on, is also a set of modular nodes and strut pieces to connect them, but made at a larger scale, designed to be easily programmed and assembled. The three iterations of the DICE project can be seen in Figure 2-1.

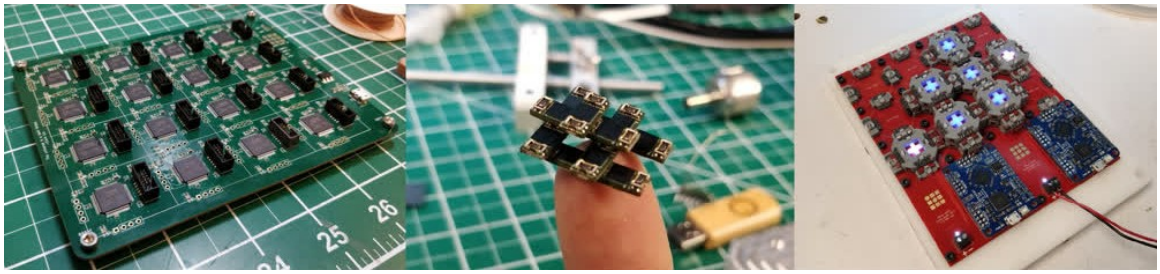

Figure 2-1: Current iterations of the DICE project. Left: Static-DICE; Middle: Tiny-DICE; Right: Meso-DICE. Images courtesy of Zach Fredin.

Each of these iterations of the DICE project are created with the purpose of im-

plementing a distributed computing architecture, with the function of breaking down any computing task into a parallelized one on the discrete hardware. They excel in certain applications that benefit from parallel computations, many of which are discussed in this work, but are meant as an all-purpose prototype to eventually become an alternative for modern computing structures. One huge benefit of this discretized architecture is the ability to align the software and hardware together. As the computations are parallelized, the nodes can better spatially represent what is being conducted within the software, which is especially true in specific applications. The processors embedded within each node depend on the scale and desired performance of each of the iterations, as do the interconnect design. Because this work focuses on the Meso-DICE iteration, a deeper explanation of Static-DICE and Tiny-DICE are not included here.

Meso-DICE consists of three main components: node, strut, and build plate (see Figure 2-2). The heart of each node is a SAMD51 microcontroller from Microchip, which is wired to a custom-made PCB that routes power, ground, and the various pins required for communication with the nearest neighboring node in each direction: north, west, south, east, top, and bottom. The bottom of the nodes have contact pads on all four sides and the top of the nodes have spring-pin connectors to press up on the aforementioned pads. These PCBs then have milled delrin and 3D printed PLA parts slotted into them, with the PLA pieces heat staked to secure everything in place. These pieces allow proper spacing and latching capabilities in order to connect with other components. The struts also have custom-made PCBs that simply have spring-pin connectors on top, contact pads on bottom, and several wires to route signals between the two sides of the strut to connect two individual nodes together. More delrin and PLA parts are assembled and heat-staked to the PCB to complete the strut. Finally, the build plate is required to have a solid base for the nodes and struts to stack on, and also delivers power to the nodes. Connectors utilizing strut components are fastened to the build plate PCB through heat staking in between the empty spaces left for nodes, and the whole structure is bolted to a larger block of milled plastic for stability. Power is routed to the nodes through a simple screw-

terminal power block, which is easily connected to a power supply.

Figure 2-2: Meso-DICE components. Left: nodes; Middle: struts; Right: build plate

The specific build plate used for testing this project had some additional capabilities built into it, chief among them being an added section for an FPGA to be soldered to, which would then be routed to each of the nodes so that it can read the data that is being passed between nodes. This functionality has proven beneficial in other aspects of the project overall, but is not put in use for the work at hand, and will therefore not be utilized.

To program the nodes, a specialized programming stand was created by adding spring pins, spacing rods, and a 3D printed platform to a custom-made microcontroller PCB. Each node has a specific pattern of contact pads on the bottom that only line up in a specific orientation (the "top" of a node is the side with the white strip, and the "top" of the programmer is the side with the notch in it). By pressing a node onto the stand in the correct orientation, it can then be programmed. This specialized stand was specifically created with robotic assembly of nodes and struts in mind. End effectors on the robot arm can grasp a node, move it to the programming stand, press down and program, then move it to the build plate and press it in place. The programming stand can be seen in Figure 2-3.

As mentioned before, each Meso-DICE node has connection pads that carry various signals throughout each node and in between nodes. A simplified representation of these sets of pads is included in Figure 2-4 and Tables 2.1 and 2.2. When any given set of pads on the top side a node is oriented upwards, you can see that the footprint is the same no matter the direction. On the bottom side of the nodes, this footprint

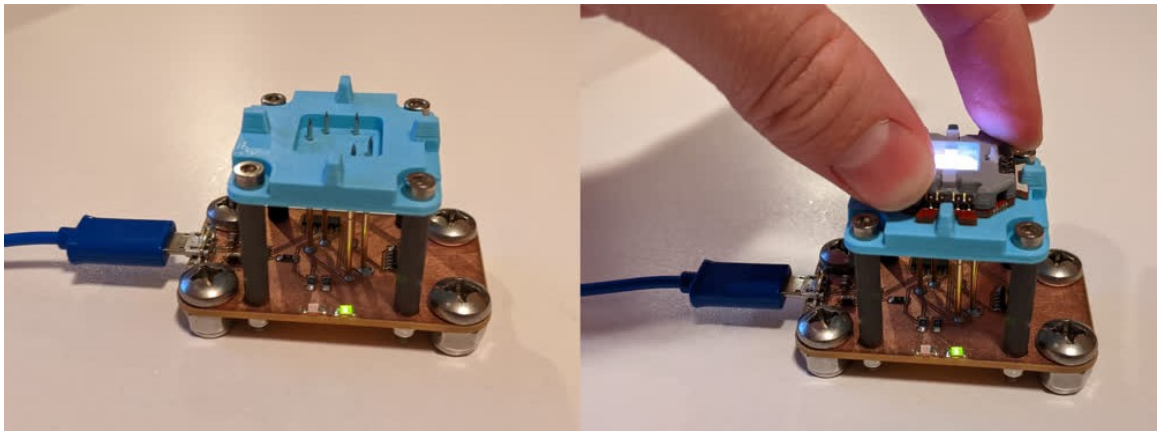

Figure 2-3: Meso-DICE node programming stand. Left: inactive; Right: programming a node

is mirrored so that signals can be ported easily from the top side to the bottom side through vias in the PCB. So, while the symmetry makes signal routing and tiling nodes together much easier, it can still be confusing to figure out exactly where each signal line can be accessed.

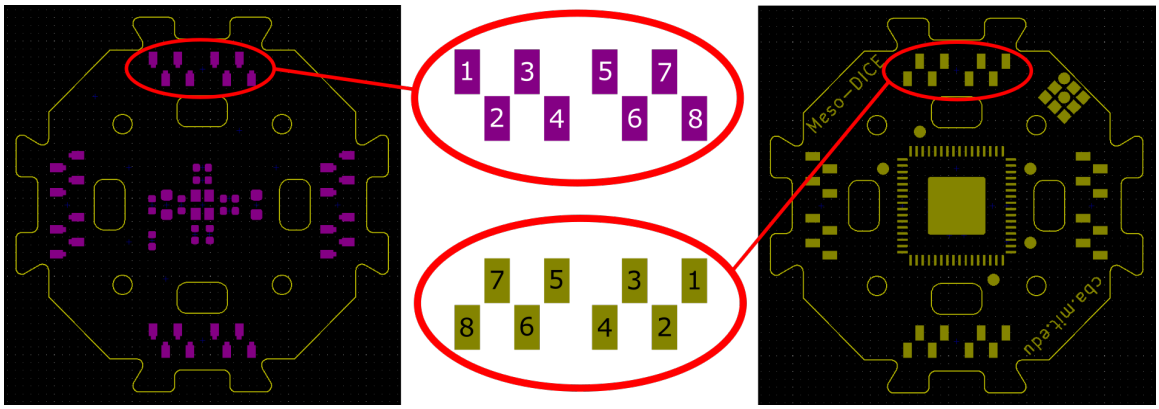

Figure 2-4: Meso-DICE node pinout diagram. Left: top side of node; Middle: zoomed in view of sets of both top and bottom node pads; Right: bottom side of node. Note that the pads and corresponding pin numbers appear mirrored to help demonstrate the connection between top and bottom pads. Meso-DICE node schematic designs courtesy of Zach Fredin.

As was mentioned previously, Meso-DICE was envisioned as an iteration of the DICE project that could be designed for end-to-end demonstrations of robotic assembly and programming [12]. Issues with the other iterations of the project included the lack of reconfigurability with Static-DICE and consistent failures of connector pieces

Pin	East	North	West	South
1	GND	GND	GND	GND
2	SIG_EAST	TX_NORTH	TX_WEST	SIG_SOUTH
3	XCK_EAST	RX_NORTH	RX_WEST	XCK_SOUTH
4	TX_TOP	TX_TOP	XCK_TOP	XCK_TOP
5	RX_TOP	RX_TOP	SIG_TOP	SIG_TOP
6	TX_EAST	XCK_NORTH	XCK_WEST	TX_SOUTH
7	RX_EAST	SIG_NORTH	SIG_WEST	RX_SOUTH
8	+3V3	+3V3	+3V3	+3V3

Table 2.1: Meso-DICE node pinout (top of node).

Pin	East	North	West	South
1	GND	GND	GND	GND
2	SIG_EAST	TX_NORTH	TX_WEST	SIG_SOUTH
3	XCK_EAST	RX_NORTH	RX_WEST	XCK_SOUTH
4	RX_BOTTOM	RX_BOTTOM	XCK_BOTTOM	XCK_BOTTOM
5	TX_BOTTOM	TX_BOTTOM	SIG_BOTTOM	SIG_BOTTOM
6	TX_EAST	XCK_NORTH	XCK_WEST	TX_SOUTH
7	RX_EAST	SIG_NORTH	SIG_WEST	RX_SOUTH
8	+3V3	+3V3	+3V3	+3V3

Table 2.2: Meso-DICE node pinout (bottom of node).

with Tiny-DICE. Meso-DICE appears to get the best of both of those iterations, while minimizing the flaws of each, to make it a viable path to continue on. For the idea of integrating displays in particular, however, the nodes are in the correct size domain to easily attach a single display to each node (or at least each node on the top layer).

At this time, it is important to recognize those responsible for the DICE project's current implementation and design. Zach Fredin, a current Master's student and researcher at the Center for Bits and Atoms, is primarily responsible for the design and fabrication of all Meso-DICE hardware, including programming and testing equipment. For more information on the development of the different DICE iterations, it will be published in his thesis, being written alongside this one. [11] Erik Strand, a current PhD student and researcher at the Center for Bits and Atoms, is primarily responsible for the code and development platform used to run and program the DICE nodes, including many of the applications discussed in this work. Examples of the first applications developed for and run on DICE hardware can be found in his

Master's thesis. [24] Now, with the design of Meso-DICE sufficiently explained, it is time to go over the displays and how they're integrated.

2.2 Display Integration

The display used for this project is a commercially available TFT LCD display, specifically the Adafruit 1.8" Color TFT LCD display with the ST7735R TFT driver. [1] The microcontroller used to control the display and interface with the DICE nodes is the Adafruit Feather M4 Express. [2] This combination was selected for a couple of reasons, including ease of programming, relative size, availability, and the fact that the Feather M4 uses the same ATSAMD51 chip that is contained in each of the DICE nodes. In all honesty, however, the exact display and microcontroller are fairly trivial components of the project as a whole. The goal with connecting the display to the Meso-DICE nodes was straightforward, so there was no need to reinvent the wheel there. To be thorough, however, the connections between the TFT display and the Feather M4 are listed in Table 2.3. Also, the connected devices are shown in Figure 2-5.

TFT Display	**Feather M4**
LITE	3V
MISO	MI
SCK	SCK
MOSI	MO
TFT_CS	5
CARD_CS	N/A
D/C	6
RESET	9
VCC	3V
GND	GND

Table 2.3: TFT display to Feather M4 connection guide.

The display itself is programmed through the Arduino IDE. After installing the correct libraries to handle both the Feather M4 and the display itself, it is quite easy to control. Utilizing the Adafruit GFX graphics library, as well as the Adafruit ST7735 display driver library, manipulating pixels and drawing shapes in various colors is

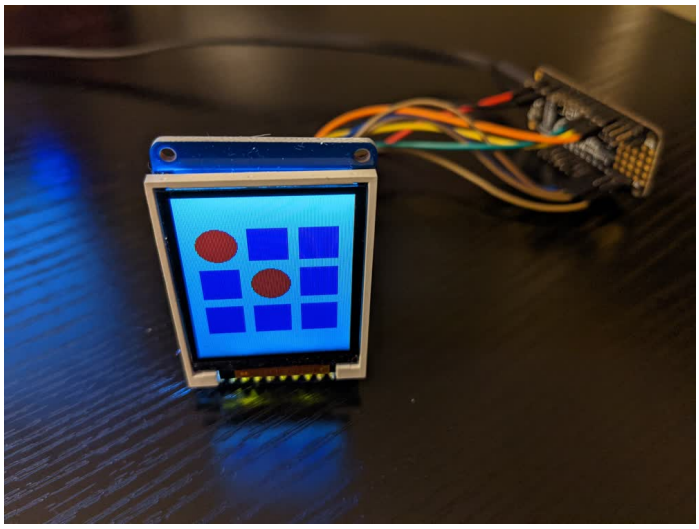

Figure 2-5: TFT display connected to the Feather M4 running some test display code.

very straightforward. Again, the wheel isn't being reinvented, so further explanation here is not necessary.

Connecting the display to the Meso-DICE nodes, on the other hand, requires some specialized hardware due to the self-contained nature of the DICE nodes and struts. A special header pin strut was fabricated specifically for this purpose. The typical PCB board used for all other struts was taken and epoxied to an electrical protoboard with wires connecting the PCB pad connections to the header pins soldered to the protoboard. This header pin strut can then be placed in the location of any other top layer strut in the structure. The contact pads on the bottom allow data to still be sent from one node to another, while also routing those signals up to the header pins, which allows that data to now be accessed by a simple jumper cable. This header pin strut is then connected to the RX pin of the Feather M4 to begin transmitting data that will be interpreted and translated to pixels on the display. This pin header strut and its subsequent pinout diagram can be found in Figure 2-6 and Table 2.4.

2.3 Sending Data

The way DICE nodes transmit and receive data between themselves is by utilizing a token passing scheme specifically created for this architecture. There are various

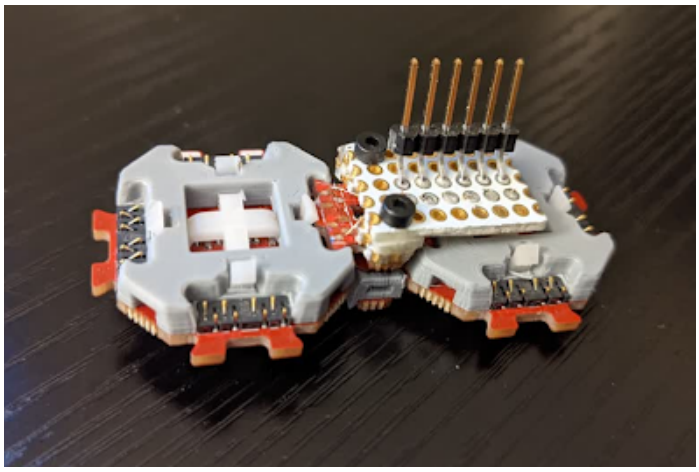

Figure 2-6: Pin header strut mounted between two Meso-DICE nodes used for signal interrogation by oscilloscope and to connect to the display.

Pin (Strut)	**Pin (DICE)**	**Signal (DICE)**
21	1	GND
20	2	SIG or TX
19	3	XCK or RX
18	8	+3V3
17	7	RX or SIG
16	6	TX or XCK

Table 2.4: Pin header strut pinout with corresponding Meso-DICE node signals.

handshakes that occur, as well as the attaching and detaching of buffers to properly manage the data passing back and forth. At first, the idea was to implement this same token passing scheme between the display's microcontroller and the DICE nodes. However, a simpler solution was to just use the Feather M4's serial line to receive specific data directly from the nodes.

By matching the baud rates between the nodes and display, the display's microcontroller can properly read data that is received from the nodes, then according to the application, determine how to translate that into what to show on the display. Testing was conducted to ensure that the appropriate data was being sent out by the nodes and properly read into the displays. For example, Figure 2-7 shows a test bit pattern being sent out from a DICE node, which was then matched with the same bit pattern being sent out from the Feather.

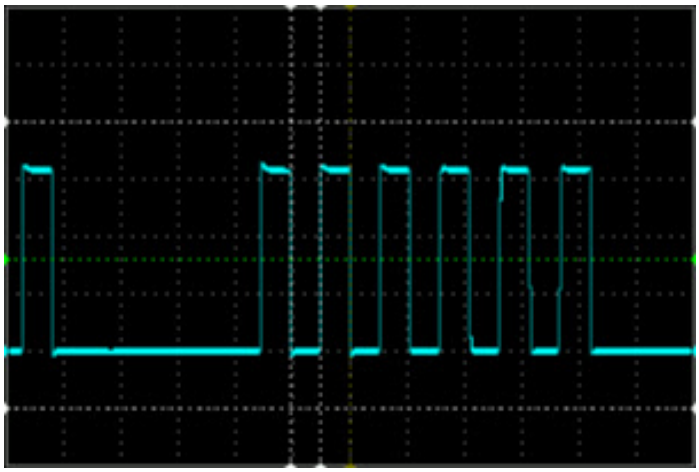

Figure 2-7: Zoomed in oscilloscope reading showing proper bit pattern being sent from node at proper frequency.

Because the data transmissions aren't happening using the same token passing scheme that the nodes typically use, we do have to use specialized code for the top layer nodes that are connected to the displays. However, depending on the application, the nodes don't transmit all of the information that would be required by the displays amongst themselves anyway, so specific top layer node code would still need to be written. Therefore, bypassing the token passing scheme for this simpler serial communication doesn't throw things off too much.

A secondary strut to intercept data from the nodes to be sent to the display was also created. By simply using some very fine 34 AWG magnet wire soldered directly to one end of a jumper cable and the pad of a strut with one set of Molex connectors removed, a solid connection was established. This secondary testing strut, as pictured in Figure 2-8 was used only for tests which required more than one display to be connected at a time. While it is definitely not as robust as the original pin header strut, it had the benefit of being connected to the TX_TOP data line, which allows the node that data is being pulled from to still have access to its surrounding neighbors. Node data needs to be properly formatted and directly sent to the pin connected to the display. If typical token passing data is sent over that line to another node, the Feather will attempt to read that data and display something according to those values. Because data between nodes doesn't necessarily contain all the pertinent

information being stored in a particular node, the Feather must opt to be sent data packaged specifically for it to interpret, rather than the smaller amounts of data being transferred.

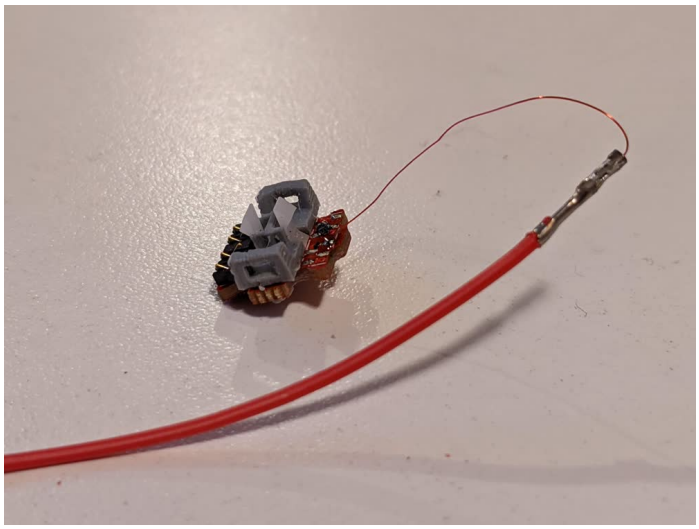

Figure 2-8: Secondary testing strut with access to TX_TOP signal line.

2.4 Packaging

Ideally, everything needed for the display could nicely fit into the same approximate volume of a Meso-DICE node and be easily connected as a top layer of the structure itself. At its current state, however, attention and efforts have been focused on determining the capabilities and unique applications that are enabled by these displays within the DICE architecture, so this ideal packaging has yet to be achieved. More work needs to be conducted to determine the best displays and microcontrollers to interface them with, fabricate Meso-DICE sized and shaped "display node" connectors with the appropriate data lines routed, and design the proper physical components needed to hold it all in place and look pleasing. This is definitely an area that needs to be explored further in the future.

Chapter 3

DICE Debugging and Data Visualization

3.1 Debugging

With the displays now connected to the DICE nodes, they can now be utilized to debug the computations being performed by the nodes. Much like print statements are used to debug code that is run on a computer with an attached monitor, the displays can be connected to any given node and have the data or any other useful information printed on the display. The nodes can either be programmed to send certain parts of the computation to the display to verify that it's working at that point in the code, or the display can intercept the data as it's being passed between one node to the other to get a sense of the interaction between the nodes (this sounds contradictory to what was stated in the last chapter, but for debugging, the specific data being passed might be of more interest than anything else).

Up until this point, there were only two ways to attempt to troubleshoot or debug code run on the DICE nodes. One way is to run the code on a simulated version of the DICE nodes. By running this code on a laptop or desktop computer, the simulation could then print out or display any useful debugging information required on the screen. This method is extremely useful in that it doesn't require any physical hardware to develop and test the code. However, there are some obvious limitations.

If the simulation isn't a perfect replication of the hardware, there's no guarantee that the code will work just as well on the hardware as it does in simulation. Sometimes there is a fault in the actual hardware that doesn't correspond to what the simulation does, so it is impossible to diagnose that problem when you don't know it even exists. By using the displays to debug the code, they can be used as a diagnostic tool to determine if there is a problem with the physical hardware or pinpoint the exact location in the code that is causing issues when run on the hardware.

The other method that has been used in debugging is logging all of the data exchanges that take place by using a connected FPGA to pull the data out of or being exchanged between each node. That data log can then be parsed for any errors and when they were encountered. The big issue with this method is that it isn't done in real-time. The data has to be parsed after the computations are completed. Errors can still be located, but not without parsing through all of the data in the whole system. Using the displays can help to see the error when it occurs and subsequently iron out that bug. Seeing it in real-time helps you know which part of the code to look at and having the display physically connected helps you even more to know which node is causing the issue.

While these other methods do have their merits and are being improved upon, the displays provide an excellent additional tool to debug any sort of code being developed for the DICE platform. They are easy to use and can show a wide variety of useful information customized to the specific needs of the application. The displays are sure to increase efficiency and ease of future development.

3.2 Data Visualization

A primary motivation of the DICE project was the various applications that it can adapt to. Some of these applications include particle simulations, machine learning, and graphics rendering. [12] Some of these applications will be explored in greater detail later on, but to illustrate the variety of use cases for these nodes and their displays, a few smaller examples will be noted here.

One concept for machine learning applications within the DICE framework is that each node represents the different input, output, and hidden layers and neurons of the system. As the nodes make their respective computations and exchange their results between each other, there is a chance that the specific structure of the neural network isn't as effective as it could be, and could require a rearrangement. Because of the modularity and robotic assembly capabilities of the structure, the nodes could be rearranged automatically and continue in the machine learning process. An idea for visualization can be easily understood by imagining that the system is learning to identify different handwritten letters or numbers. The input, in this case a specific handwritten letter or number, could be shown, as well as the paths of the interconnections between the different neurons and layers, and finally, the letter or number that the system has identified at the end. Being able to physically see the entire network working, almost appearing as to be living and breathing, is sure to add better insights and understanding to how the network is performing, as well as how it can be improved.

Another simpler application is that of parallel computing. In one instance of this, the DICE nodes work together to perform a simple calculation of pi. Each node is given a specific section of the calculation to perform and upon completion of its portion, it sends its results along the chain until finally terminating at the final node that sums each portion up to determine the final result. A display can be attached to the final node in order to see the final result, or it could be connected anywhere along the chain in order to see the information being cascaded down the line. The DICE implementation of this calculation was developed by Erik Strand.

The above example was actually put into practice as a demonstration of the visualization and debugging capabilities of the DICE display hardware. In this test, multiple chains of nodes, starting with 2 and going up until 12, were assembled and connected. As the calculation moved up and down the line of nodes, the final result was sent to the display. This demonstration and its results are shown in Figures 3-1 and 3-2 and Table 3.1. The differences in the received values for each test show that with the increasing number of nodes, the precision of the answer increases, as well.

While what is displayed isn't necessarily the most exciting thing to look at, it does show how simple it is to connect the display and read out data being processed within the nodes. The time it took for each calculation to complete was also measured. As can be seen, the addition of each node increases the amount of time it takes to complete the computation fairly linearly. Every single node is still doing the same amount of operations as the last, so the total number of operations scales proportionally with the number of nodes.

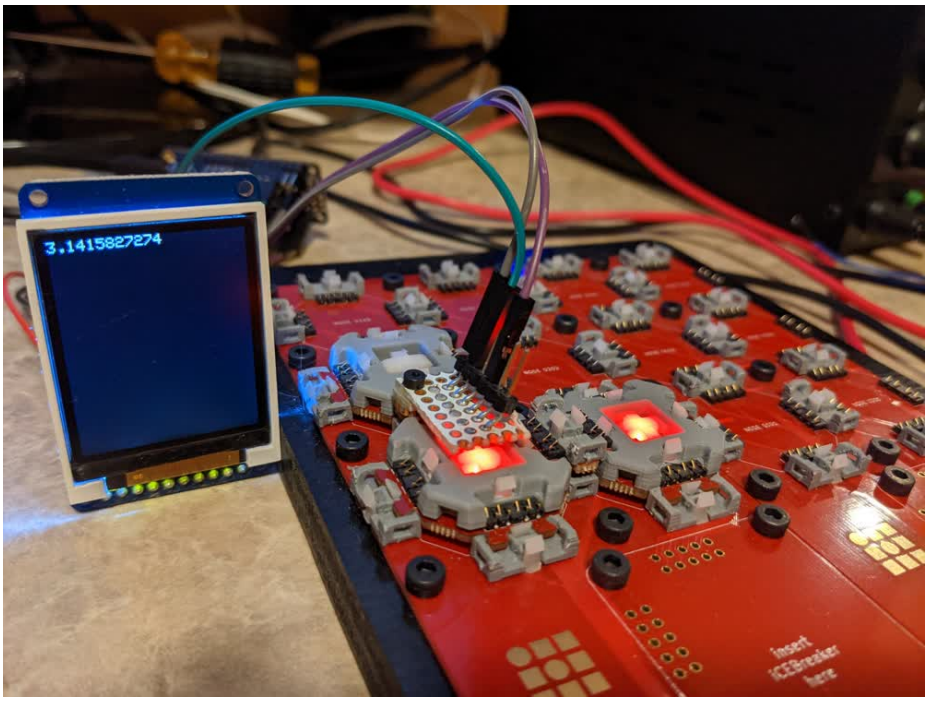

Figure 3-1: Pi is calculated across two nodes with the final result sent to and shown on the display.

In terms of graphics rendering, there are many different ways to visualize the relevant information. One idea is to have the displays render cross-sectional views of an object according to the different layers of physical nodes that represent the different spatial layers of the simulated object. Another more abstract idea is to have multiple surface layers of displays assembled together to act as faces of a cube so that each face of displays would be able to show a rendered object from its respective direction. A deeper look into a specific graphics rendering technique will be explored in the next chapter. Also, while more advanced displays would be required to give a true

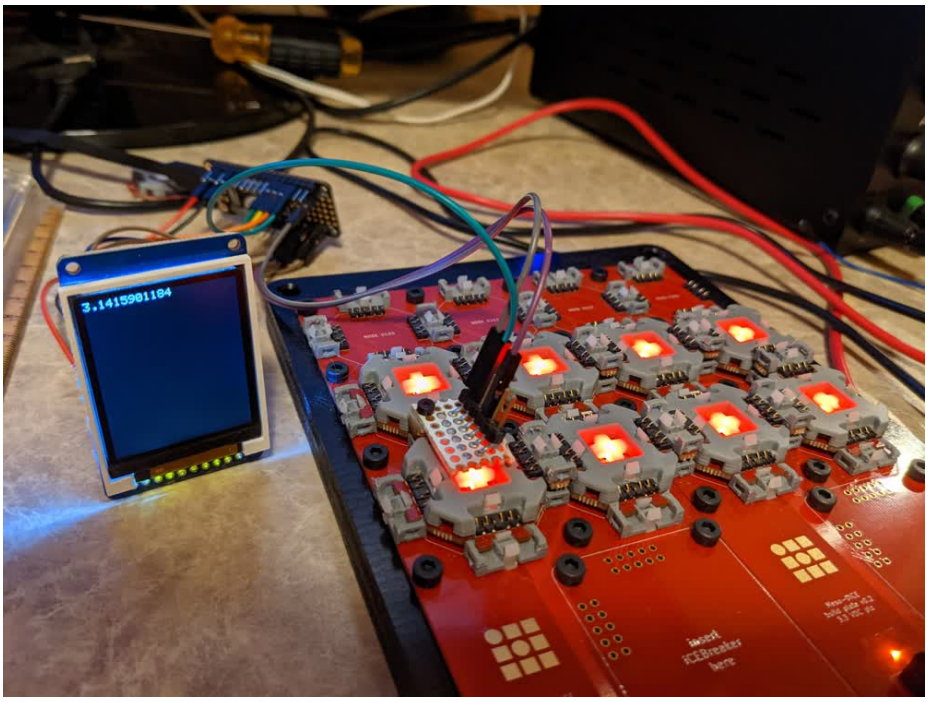

Figure 3-2: Pi is calculated across eight nodes with the final result sent to and shown on the display.

# of Nodes	Pi Calculation Result	Time Elapsed (s)
2	3.1415827274	1.1
3	3.1415860653	1.3
4	3.1415877342	1.5
5	3.1415886879	1.7
6	3.1415894032	1.9
7	3.1415898800	2.2
8	3.1415901184	2.4
9	3.1415903568	2.6
10	3.1415905952	2.8
11	3.1415908337	3.1
12	3.1415910721	3.4

Table 3.1: Pi calculation test results with varying number of nodes.

three-dimensional effect, the parallel computing power of the nodes could fairly easily compute different volumetric and holographic data that could then be showcased on a given face or multiple faces of a display structure.

These, along with all of the other countless applications that have yet to be explored, demonstrate the wide variety of new capabilities that not only the DICE

nodes themselves enable, but also the integrated displays. As this project is still so recent, more development of DICE applications will be needed in order to continue maximizing these capabilities. It is hoped that others will see the usefulness of this technology and desire to develop and test their own applications and subsequent visualizations using the platform. While the exact effectiveness and value that these displays add will be quantified and discussed later, it is apparent that this could be the foundation for a whole new way to approach computing and visualization.

3.3 Code Example

Revisiting the pi calculation example, a specific adaptation that is needed within the code should be discussed. While the specific method used may seem simple, it proved to be quite the non-trivial development in getting both hardware devices (node and display) communicating properly with each other. Within the DICE code, there is an object called "token_payload" that stores that individual node's calculated pi value, summed with what was shared with it from the next node in the chain. This value's type is a float, but the DICE hardware was developed to transmit uint32_t type data. A simple data conversion here isn't possible due to rounding errors causing a complete loss of precision. Therefore, the "reinterpret_cast" function is utilized with a combination of pointers and pointer addresses to preserve the data for the transmission. The data is then sent over one of the node's TX lines to the Feather for processing. This whole process can be seen in the following code snippet:

```
// Prepare the data that will be sent
float pi_actual = 3.1415926536;
float pi_result = token_payload->pi[index];

// Convert it to a uint32_t to send
uint32_t displayData[2];
displayData[0] = *reinterpret_cast<uint32_t*>(&pi_actual);
displayData[1] = *reinterpret_cast<uint32_t*>(&pi_result);
```

```
// Send both values over the TX_NORTH line
tx_north(displayData, 2);
```

On the receiving end, there is a bit of work to receive the entirety of the uint32_t data, in the form of reading in each byte individually and then concatenating it all together. However, once it is received and processed correctly, it can then be converted back into a float using the same "reinterpret_cast" function, but replacing the variable type specified. Some tricks are then implemented to erase the previous value from the display before finally printing the received value to the display.

```
// Make sure the whole data packet is available
if (Serial1.available() >= 4) {

    // Read in the data one byte at a time
    byte byte0 = Serial1.read();
    byte byte1 = Serial1.read();
    byte byte2 = Serial1.read();
    byte byte3 = Serial1.read();

    // Combine the bytes into the whole uint32_t data
    uint32_t incomingData = (byte3 << 24) | (byte2 << 16) |
        (byte1 << 8) | (byte0);

    // Convert the uint32_t data into a float
    float piData = *reinterpret_cast<float*>(&incomingData);

    // Erase the previous result from the display
    tft.setCursor(0, 0);
    tft.setTextColor(ST77XX_BLACK);
    tft.println(prevData, DEC);
```

```
    // Print the new result to the display
    tft.setCursor(0, 0);
    tft.setTextColor(ST77XX_WHITE);
    tft.println(piData, DEC);

    // Update the prevData value (same type as piData)
    // Note: prevData is initialized outside of this function
    prevData = piData;
}
```

Note that in this example, there would be two values being sent over and received, the actual value of pi and the calculated result. However, only the calculated result would remain long enough on the display to actually be able to see it. It's trivial to implement more code to keep both values visible, but this code snippet is meant as a simple example of how data can actually be transmitted from node to display. With that simple demonstration out of the way, we now move on to a more complex example.

Chapter 4

Application: Particle Simulation

4.1 Motivation

As the DICE project was inspired by the CAM-8 project, it only seems right to tackle one of the main issues that system was built for: particle simulations [20]. The way one particle interacts with another can be pretty straightforward to simulate, but once you add in hundreds, even thousands of particles into one system, the simulation can get bogged down pretty easily due to the strain on the computing resources. However, when applied within the CAM-8 or DICE architectures, the parallel and locally driven computing processes make such a task much more feasible. From simply determining how a set of gas particles will move about in a given space, to analyzing the strain experienced by particles in a complex structure under duress, physical particle simulations can provide a lot of beneficial information before a physical experiment is conducted. Being able to run such a simulation without needing the pure processing power of something like a high performance computing center is quite the alluring prospect.

Rather than just running these simulations and checking the results after the fact, however, it is much more informative to see the full simulation running in real time. Without that capability, there is a possibility of missing out on vital information, such as the exact interactions that happened before a strain-induced fracture occurred, or to be able to notice an overall trend or pattern that would have been much harder to

decipher with a more general look at numbers on a page. For all of these reasons, it is desirable to implement a structure for performing particle simulations within the DICE display framework.

4.2 Implementation

A simple particle simulation can be construed where each particle has simple interaction forces with each other particle, and the system on a whole can be observed in its movements. As mentioned before, each DICE node is given a specific region of space which it is responsible for. Any one particular node will compute the interaction forces and trajectory for each given particle within its spatial boundaries. When a particle's trajectory is headed for a boundary between nodes, the data for that particle is then sent to that neighboring node, which then assumes responsibility for that particle as it simply crosses over between nodes and their respective spatial boundaries. If a particle approaches a boundary without an adjacent node, that boundary is treated as a wall which the particle will just bounce off of and continue within that region.

A few different versions of this sort of particle simulation have been developed for use with the DICE architecture, all by Erik Strand. The most robust example of these is that of a simulated DICE node environment that utilized a grid of virtual nodes, each running the same general simulation code, then displayed the results of the simulation on the screen. [24] This simulation successfully proved that the ideas behind DICE node interaction worked and that the data could be successfully computed at the individual node level, then passed on to its appropriate neighbors to correctly model particle behavior. While this particle simulation performs exceptionally well, it is only virtual, and has not been directly ported to code that will properly run on the physical DICE nodes. Much of it has been adapted, however, and recent results show the simulation running properly on a single Meso-DICE node.

It is that current iteration of the developed code that the particle simulation with display functionality is tested. Using that version of the code to guide the implementation, a demonstration was created that would run the single-node particle

simulation and after each step, send the location of each particle to the display, which would then update continuously to show the movement of the particles. Similar code adaptations to those presented in the previous chapter in the pi calculation example are used to pass along the precise location of each particle, as well as to erase and redraw the particles according to their new positions.

The display is first initialized by drawing a simple bounding box square to represent the boundaries of the simulation. This can be adapted as needed or desired, but a simple box of size 100 pixels by 100 pixels seemed appropriate. Then, like the pi calculation example, the display starts listening for the particle information to be sent over. Every time that a new set of data is sent over, it reads and interprets the data, then draws a small point at the particle's location, mapped to the display size. This data that is read into the display comes in sets of x- and y-coordinates for each of the particles and is therefore quite simple to store and parse. The simulation code is then set on a loop so that the DICE node continuously updates the positions of the particles according to the velocities of the particles, then sends that positional data to the display so the simulation visualization can be continuously updated.

While simple, this setup can be used with even more complicated particle simulations, including when node to node communication is implemented fully on the hardware. The display code just expects its connected node to update it with particle positions, and will update the display accordingly. So long as the nodes send that data over, it doesn't matter what the interaction forces with other particles, boundaries, or whatever, the simulation can still be successfully displayed.

4.3 Results

After working through some bugs in the initialization of the display, the first particle was able to be seen on the display. It moved about fairly slowly (that particular test particle has a relatively low velocity), but there did not appear to be any glitching or sudden movements, and the display updated quickly enough for a smooth picture. As more particles were added to the simulation, the display was able to show each of

them moving around smoothly, as well. Faster particles appeared fast, slower particles appeared slow.

In terms of the DICE code, each of the particles appeared to obey the laws of the interaction forces given to them and to appropriately bounce off of any node boundaries present. As it was a very basic simulation, no unexpected or erratic behavior occurred. The code and algorithms run on the DICE node has proven to be robust, but that is of less interest to this test than the performance of the display.

Rather than seeing strings of numbers continuously updating over the serial monitor (which happened often in the initial development of the node to display code), it was much easier to determine that the simulation was working in its entirety when it was fully visualized. Each particle can be easily isolated and tracked. Interactions between particles and the boundaries could be clearly seen. Figures 4-1, 4-2, and 4-3 show the simulation and subsequent visualization running at various particle counts. It all was a great showcase of the merits that the display bring to these simulations overall.

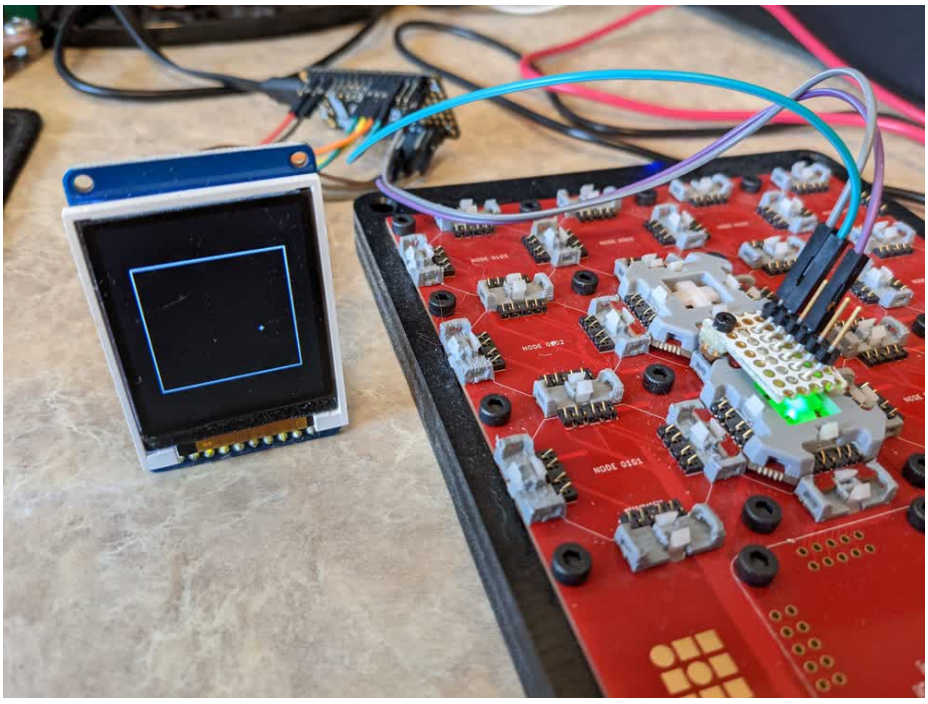

Figure 4-1: Particle simulation being run and visualized on the display with one particle.

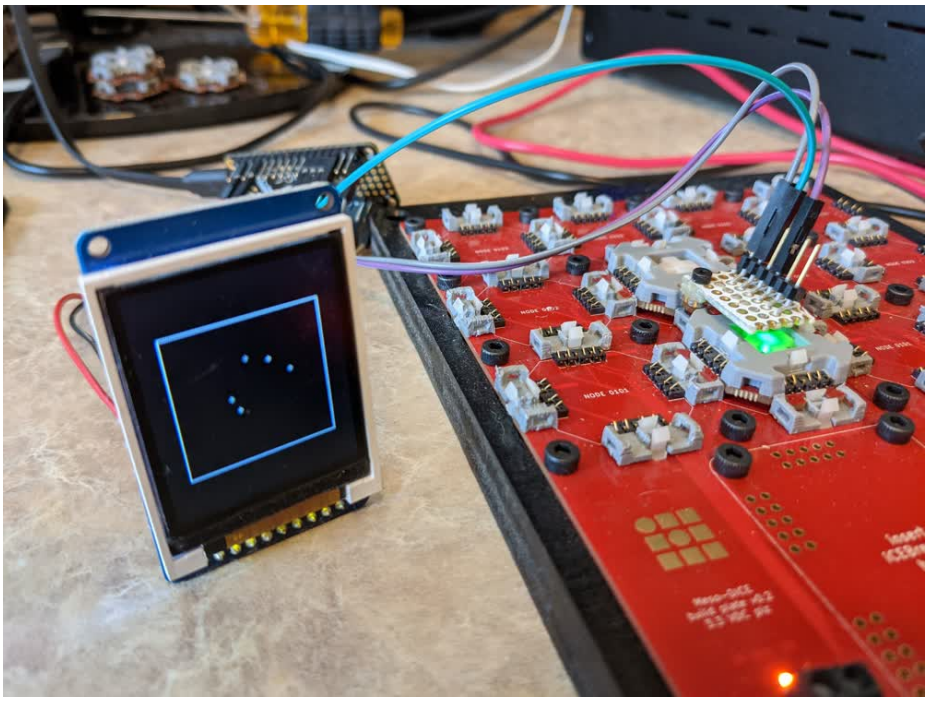

Figure 4-2: Particle simulation being run and visualized on the display with five particles.

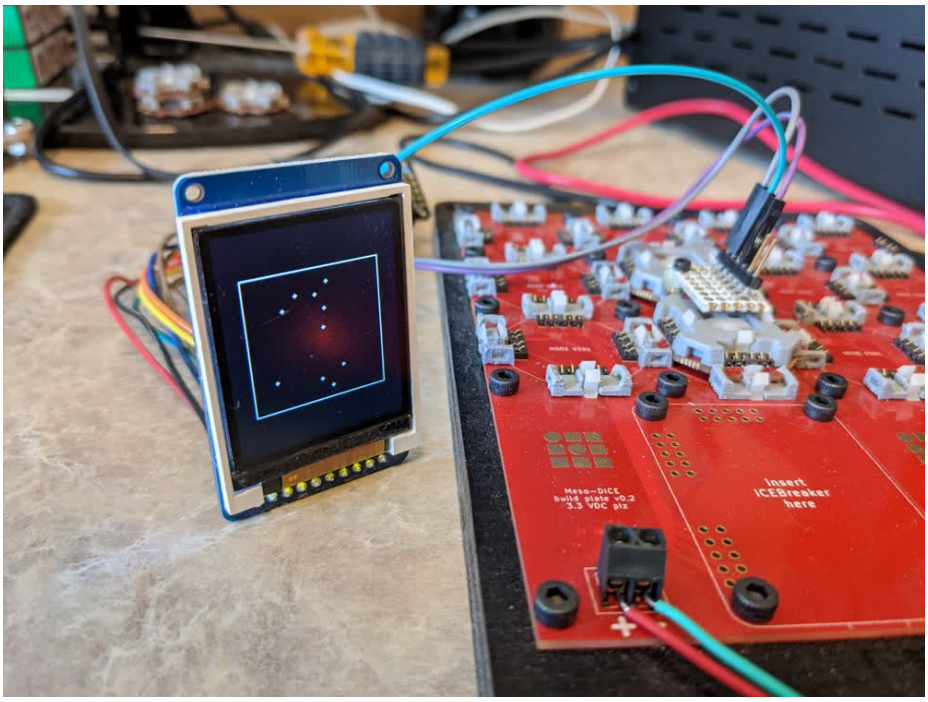

Figure 4-3: Particle simulation being run and visualized on the display with ten particles.

However, that's not to say that the visualization was absolutely perfect. The main area of concern with the performance of the display is the speed at which it's able to update, and if that's fast enough to be a satisfactory visualization. If the node is calculating faster than the display can receive, interpret, and display data, then the data won't be organized properly and is therefore useless for the visualization. In most applications, there will be many more computational steps for the nodes to compute than the displays, so sufficient time should be allowed between display updates.

Unfortunately, this was the case in many of the tests, and the node's simulation code needed to be slowed down in order for proper communication and organization to be maintained. The simulation code already had a function it would call to wait so many clock cycles before running the loop of updating positions and velocities again, but that amount of time wasn't enough depending on how many particles were present in the simulation.

In fact, to determine the performance of the display in the simulation, a simple test was conducted to see how big of a delay (how many clock cycles to wait) were needed in order for the display to reliably keep up with the simulation, all while varying the number of particles present in the simulation. The results of this test can be found in Table 4.1.

# of Particles	**# of Clock Cycles**	**Delay Time (ms)**
1	100000	2.083
2	100000	2.083
3	150000	3.125
4	230000	4.792
5	250000	5.208
6	600000	12.500
7	350000	7.292
8	800000	16.667
9	520000	10.833
10	570000	11.875

Table 4.1: Delay time required per particle for display to reliably update all positions without the data getting unorganized. Delay time based off of DICE node's 48 MHz clock cycle.

As can be seen, as more particles are introduced to the simulation, the more time the display needs to process all of the positional data and update the display. At lower particle counts, the delay isn't very substantial, but that is not the case as more particles are introduced. While the inherent slow feeling could be attributed to some of the test particles having relatively low velocities, the delay does become quite noticeable, regardless of the speed at which the particles are moving. What is really interesting is the two outliers in the measurements. For every particle count, the delay time increases slightly (or stays the same in the case of 1 and 2 particles). However, for a particle count of 6 or 8, the delay time increases substantially. This trend was validated on more than one node, and is quite puzzling. There doesn't seem to be any major differences in the code for those particle particles in terms of their initial positions and velocities, yet the delay required for them to run properly is quite erratic. More testing will need to be conducted to figure out what is going on in those instances.

4.4 Improvements and Variations

The first obvious improvement that needs to occur is increasing the speed at which the display can operate. This must be done by optimizing the code and improving any glaring time-consuming functions that cause everything to take so long. More of the processing of the data could be offloaded to the the nodes themselves, but it still needs to be converted, sent, converted back, then displayed. More tests would be required to determine if the display's microcontroller or the DICE node is faster at certain tasks in order to determine exactly what needs to be done where. The whole point was to access data already being stored within the DICE node and make a compelling visualization out of it. One final more drastic solution would be to completely revamp how data is shared between node and display. For example, other communication protocols, such as I2C or SPI could be used, which can be faster than the UART protocol implemented throughout this work.

For the single-node simulation, all particles are located in the same node's spatial

region, and the node has wall boundaries set on all sides. In theory, the multiple-node simulation could go in one of two directions. Each node can either be responsible for a specific set of particles to keep track of over the whole simulation, or they can represent the specific spatial region and keep track of any particles that enter that region. The latter method makes it easier to scale the simulation, however, so that is the route that will be taken for the application.

Therefore, next obvious thing to do is to continue developing the code so that it is capable of fully simulating particles crossing boundaries between nodes using the necessary node to node communication there. It has been shown to work on virtual instances, but not yet on the hardware. The display code should still work by displaying whatever particle position data it is fed, so development would be more on the DICE side of things. As a quick concept of what such a test could look like, Figure 4-4 shows two nodes running the simulation next to each other (but not technically passing data between each other), with two separate displays visualizing the particles from their respective node. The simulations were started at different times to show some variety in particle positioning. Also, the green lines of the boundary boxes are representative of the shared boundary between the nodes. Again, this is just a concept of what it could look like, especially in terms of scaling up the simulation, but more work needs to be done.

There are a variety of other more complicated visualizations that can occur, all possible because of the simplicity of the parallel calculations and data transmissions between nodes and to the displays. For example, colors can be applied to the various particles to represent relative velocity to one another. More complex interaction forces can be introduced with different signifiers of different physical phenomena being coded into the visualization based on variations in size, color, speed, etc.

Another feature than can be implemented is by utilizing the third dimension that the DICE nodes are able to stack in and transmit and receive data. DICE nodes can be stacked on top of one another, allowing for a 3D particle simulation. One way this could work is to have all nodes within a stack of nodes send the coordinates for each particle in their region up to the above node. Once all coordinates for each particle

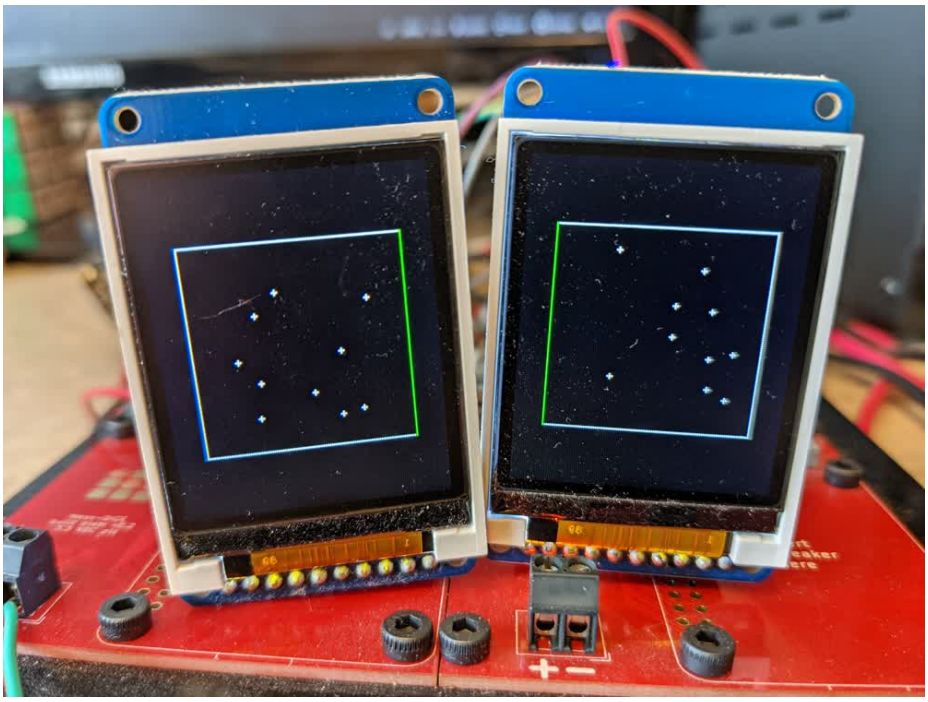

Figure 4-4: Concept visualization of what a particle simulation with two nodes communicating could look like. The green lines are representative of the shared border between the nodes.

in that stack of nodes is collected by the top layer node and sent to the display, the varying depths of the particles could be easily visualized using a simple shader. For example, the closer to the top, the more opaque the particle, and the closer to the bottom of the stack, the more transparent. As particles move higher and lower within the stack's spatial region, the corresponding point on the display can get more or less visible.

The two simulations run for this demonstration are in both two and three dimensions. The 2D simulation is a 4 x 3 node area, and the 3D simulation is a 3 x 2 x 2 node volume. Both simulations contain the same number of particles, with the height, width, and depth units being equal. The particles have just the basic interaction forces present, allowing them to bounce off of one another upon collision. Each particle is given a random initial velocity to incite movement and interaction.

One final thing worth mentioning is that of the characteristic of the displays that they currently work according to how well the data they receive matches up with the data that they're expecting. This can absolutely be both a bug and a feature,

depending on how you look at it. If mismatched data shows up, it breaks everything and ruins the visualization. On the other hand, though, it allows the visualization to be customized specifically for the application that it is working with, and can also help debug the problem of figuring out why the mismatched data occurred. It's a two-edged sword there, and there's sure to be more developments with it as this platform is used more and for more applications.

Chapter 5

Application: Ray Tracing

5.1 Motivation

In the world of graphics rendering, the next iterations of the highest end commercially available products are locked behind years-long multi-billion dollar development cycles. These Graphical Processing Units (GPUs) specialize in delivering the best graphical quality for whatever sort of application they're applied to, being able to handle large amounts of data that make up the polygons, meshes, textures, colors, and everything else that is involved in a given scene or simulation environment. Like other high-end modern day computing products, they rely on an architecture that doesn't build in locality from the foundation, but still do extremely well with the extremely specialized and optimized architectures that they do implement.

Trying to build a GPU using DICE nodes and displays is an idea based on the desire to overcome this time consuming and expensive iterative process. Taking advantage of the scalability of the DICE architecture, a more complex or higher performing GPU could be achieved simply by scaling up and adding more nodes. For an application that requires just a little more or less rendering power, this flexibility in scaling could introduce a solution to providing the most efficient needs-based GPU for each application. It wouldn't be necessary to wait for the next, more powerful iteration of a device or get more power than is truly needed. It is this idea that makes DICE an attractive solution for graphics rendering applications.

In particular, however, there is one graphical application that stands out from the rest: ray tracing. Ray tracing is the idea of computing the optical paths that each ray of light in a scene will travel in order to determine the exact lighting, color, and reflections that would naturally occur from that object. The general idea was first introduced by Albrecht Dürer in 1525, [14] but has been worked on extensively over the past few decades as the field of computer graphics has grown substantially. Many companies in the industry have developed all sorts of sophisticated software and programs to simulate and calculate all sorts of computer generated scenes, particularly those involved in making animated movies and effects. [7]

Because ray tracing algorithms are so computationally expensive, use cases for them have typically been limited to those of static images or pre-rendered scenes. However, advances in the hardware and in the algorithms themselves have led to enabling of real-time ray tracing capabilities. While it would seem that devices capable of real-time ray tracing would be limited to the most cost-prohibitive and specialized equipment, the technology can be found in the most recent generation of home video gaming consoles. However, even in those instances, extremely specialized hardware is used and took years of research to attain.

Therefore, it would be greatly advantageous to implement such a complex application on the simplified DICE hardware and use its parallel processing power to handle the heavy computations required. Utilizing the DICE architecture, the complex ray paths that help realistically render a scene optically could be divided up and computed locally to share the load and not be limited by overall memory speed and size. There are some limitations, such as resolution, that can arise from not using the most high-end chip-sets for the local computations, but the size and complexity of the scene shouldn't be an issue as the work can continue to be divvied up and executed in parallel by the entirety of the system.

The greater implication of executing such a system is, as was mentioned, the scalability of the system. By adding more nodes and displays, the resolution of the rendered image, the complexity of the scene, and the ability to perform it all in real-time is enhanced. The fundamentals that will work on the most basic of demon-

strations can feasibly be made to work on a much larger system. The scaling iterative systems that could be created could combat the aforementioned years-long development cycle, all without having to build everything from the ground up. For these reasons, this application has been conceptualized and will hopefully help overcome these challenges.

5.2 Proposed Implementation

Before digging in on the exact implementation of this application, it is necessary to note that at this point the threshold of what has been achieved on the DICE hardware has been crossed. No one has yet attempted to create a ray tracing simulation on the system yet, either virtually or physically. Due to time constraints and the need for further development of the DICE node and display code, it is necessary to approach this application from a theoretical point of view. As more progress is made on the development front, the transition from theory to actual experiment should be quickly realized.

The implementation of this attempt at locally computed ray tracing takes advantage of the duality of light in that it acts like both a wave and a particle. Essentially, this application would utilize the same framework established by the particle simulation methods found in the previous chapter, but the particles are instead replaced with photons and the interaction forces between the particles are replaced with equations that dictate optical forces and phenomena. Rather than a particle simulation, this would transform into a sort of "photon propagation" simulation.

The DICE nodes would represent a volume which can be filled with various objects to be depicted by the displays. As the photons propagate through the scene, the rays will reflect off the object and those interactions can be calculated and recorded. The top layer of the simulation volume will act as a viewing window that captures the depiction of the scene based off of the interaction between the photons and the objects.

It's easy to think of ray tracing as "tracing" each ray of light as it is emitted from a light source, reflects off of various object surfaces, then exits the viewing window

where the viewer's vantage point is located. However, in practice, this method is not very feasible because it requires keeping track of every single ray of light emitted from the light source. So, in most algorithm implementations, the calculation starts at the vantage point of the viewer, goes through the viewing window, then goes on to interact with the various surfaces. Any rays that do not interact with an object or light source are represented by darkness (or the chosen background color). The rays that do interact with the objects or light sources then have the color calculated based on the object and light interactions, and that color is then displayed on the portion of the viewing window that the ray "originated" from. This typical method of ray tracing is called "forward" ray tracing, while following the actual path that the light would travel, from light source and eventually to vantage point, is called "backward" ray tracing. While it was mentioned that it's not very feasible to use backward ray tracing, it is necessary to simulate the effects of diffuse reflection of indirect light. [4] However, this is beyond the scope of this implementation, for we are just aiming to get basic ray tracing algorithms working within the DICE architecture.

Concept art of what this all might look like within the DICE perspective is included in Figure 5-1. The entire surface layer of displays (or just one display) will act as the viewing window through which the rays from the vantage point are sent. Each node will represent the volume of the scene, including representations of objects and light sources. Depending on the resolution used for the calculation, the corresponding number of photons will begin to propagate, starting at their own local node, then moving throughout the volume, being transferred from one node to another, just like the particles do in the particle simulation. When a given photon comes across an object, the proper reflection or transmission is calculated based on the characteristics of the object's surface at that point, then it continues on. Based on the paths and interactions, the appropriate color is reported back to the portion of the display(s) that the photon started from, which all combine into one all-encompassing image.

Note that this concept is applicable to all forms of ray tracing, not just the potential DICE implementation. The concept art is therefore very similar to the typical diagrams found in any medium with an explanation of how ray tracing works. What

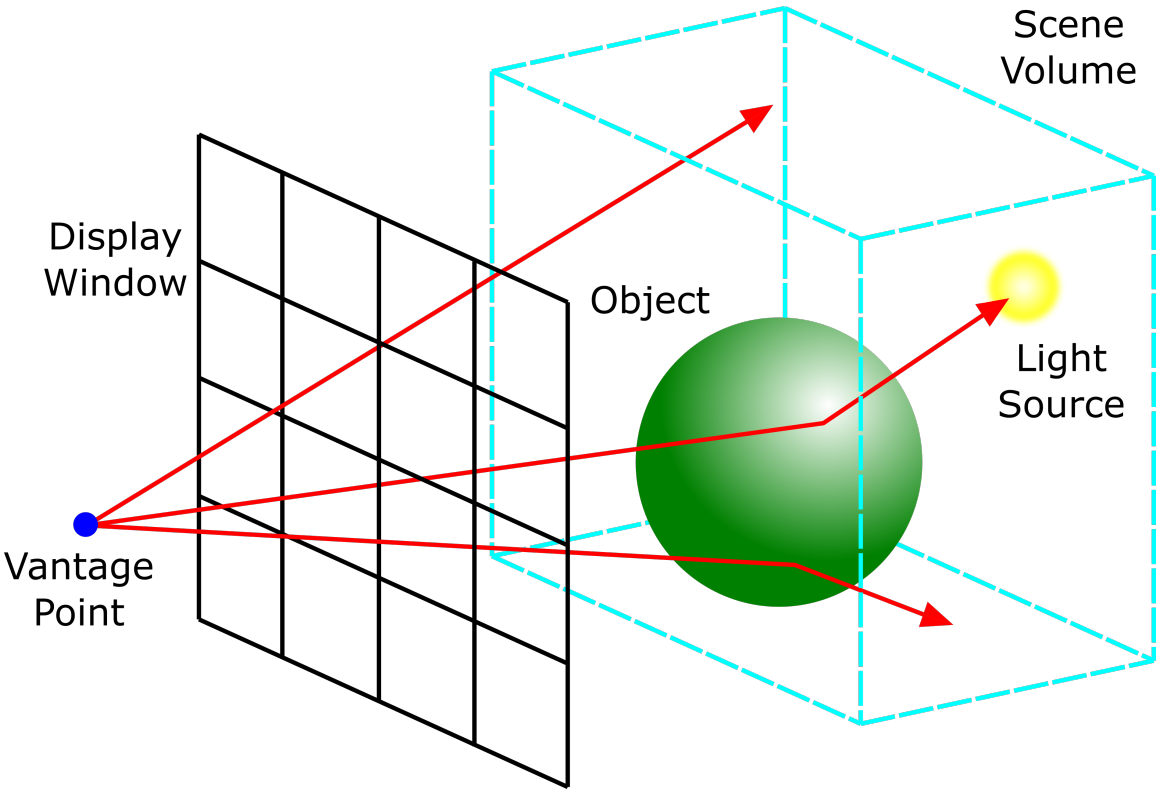

Figure 5-1: Concept art of the DICE ray tracing implementation. The display window in black represents pixels of a given display or a wall of full displays. The dashed cyan box represents the scene volume that will be broken up among various nodes to compute the paths of the rays. The red arrows show the paths of those photons as they propagate through the system, originating computationally from the blue vantage point. Objects and light sources (green and yellow spheres, respectively) will be represented in the scene volume.

sets this concept apart from all those other implementations, however, is the discrete methods that are employed within the DICE architecture. The volume containing the scene is broken into discrete spatial chunks that keep track of the photons propagating throughout the scene. The resulting rays that are traced and their determined colors are then represented on the displays at the point where the rays would have passed from the vantage point through the viewing window. This sort of "discrete ray tracing" has yet to be implemented, and the DICE architecture seems to be more than capable of achieving it.

Because the initial particle simulation worked on the virtual instance of the DICE hardware, [24] it suggests that by building upon that basic framework, this proposed

implementation should work, as well. Assuming success, this implementation would then have the capability to be scaled to create even larger displays or to increase the resolution or processing power of the computation. Each node has a certain computational capacity and by dividing up the work and with enough nodes, any heavy computation can theoretically be handled. There is obviously much work to be done to actually implement this within DICE. First off, finishing the development of the particle simulation to work on the physical hardware will help significantly as it builds the photon propagation framework that is required for this application. Next, adapting the optical equations into the code would allow for realistic path tracing to occur. Creating the objects and their surface characteristics is also a big part so that there is actually something to be reflected off of and rendered on the display. Simple tests with very simple objects will be the way to go once these different aspects are handled properly. From there, the viability of the DICE implementation of this application can be measured.

Beyond that, there are a variety of other ray tracing techniques that could be implemented to handle more complex scenes and ideas, such as distributed ray tracing to handle problems such as motion blur and fuzzy reflections. [8] Another technique that would be worth looking more into is that of "parallel rendering," which is essentially what we're already doing by breaking up the rendering tasks for each node, but more intricate and detailed methods exist. [22] Regardless of the complexity, for any further implementation, it would be wise to consult the plethora of information and guidance provided by the experts in the field. [18, 3]

Chapter 6

Evaluation

6.1 Debugging Enhancement

As was demonstrated in an earlier chapter, the ability to connect the display to any of the top layer nodes proves to be extremely beneficial when it comes to debugging the code running on the DICE nodes. Not only does it allow for detecting the problem in real time and determine which step of the code the issue occurs in, but it also pinpoints the exact node where the problem lies. While it's not as streamlined as using the virtual DICE nodes to emulate the code, when used in tandem with that method, code can be quickly developed and verified on the physical system that it will be running on when all is said and done.

Quantifying such an enhancement is a bit of a tricky task, but it can still be discussed. Much like having certain points in the code where a program can pause, verify the correct data, then continue on, the displays can act as such a troubleshooting diagnostic tool. Without actually having DICE code developers use the displays to test out its debugging enhancements, it is hard to determine exactly how much this tool will be utilized, but for the sake of this work, it has been of great use.

At its current state, debugging can be a little frustrating due to the fact that the node needing to be reprogrammed must be removed along with the pin header strut and other struts connected to it. Upon reprogramming, the structure must then be reassembled before more troubleshooting can occur. While this isn't necessary an

added issue because of the displays, it still can be time consuming. One thing that can overcome this is the end-to-end programming and assembly process that is being developed for the system using robotic assembly, however that may not be practical on an individual development basis.

Finally, as improvements in the packaging and optimization of the data transmission processes occur, the ease of use of the displays as a diagnostic tool will only increase. At that point, hopefully many more applications will have been explored and tested, giving many more opportunities for the refinement of the system and its effectiveness. At its current state, however, it does seem to be a worthwhile addition and contribution to the project overall.

6.2 Data Visualization

The DICE nodes on their own are capable of so many interesting applications and computations, and the addition of the displays into the system only amplifies the usefulness and intrigue of them. Being able to observe the workings of a system proves to be invaluable in gaining a better understanding of how each process unfolds. The amount of unique ways to showcase the data is seemingly limitless and can therefore elevate even the most mundane of measurements into more interesting observations that are more likely to unveil useful patterns hidden within the data.

There are some things to be aware of, however, when adding all of these visualizations to a given application. By adding more things for the system to do, it does tack on an additional time cost. While the system is still able to work asynchronously, the displays must be able to keep up with the data that the nodes are sending. If they don't, the data won't match up or the whole program must slow down to allow for the displays to catch up, making some applications less useful. This may not be an issue for all applications, especially for those with complex computational steps, but it can't be ignored outright.

It's also worth noting that some applications benefit more than others from having the displays in the first place, or some need only utilize one or two displays for the

beginning and end of a chain of data, rather than the entire area of display space. The value that the visualizations truly add is quite subjective, and can really only be determined according to the developer and observer of whatever applications are run. All in all, there are some drawbacks to be aware of when using the displays and they must be considered in comparison to the benefits provided by them.

While the data that is shown on the displays is reliably correct, as in the displays only show exactly what is being read in and interpreted (which is why the unique "reinterpret_cast" function discussed in Chapter 3 is needed), the connections between the physical display and node, as well as the connections between the nodes, can be suspect at times. This causes incorrect behavior from the visualization, nullifying the usefulness of the visualized data. These unstable connections appear to be caused by iterative wear and tear from repetitive attachment and detachment from the build plate and other components. A sort of pre-loading structure to press down on each of the nodes and struts has been discussed, but is just one potential solution. This only seems to be an issue when more nodes are added together.

The added usefulness of the data that is visualized makes the integration of the displays a necessity for most applications. The cost to add the displays can be reduced by finding a display that's better suited for the DICE nodes in terms of size and design integration. More work will go into optimizing the data transferring to improve the speed of the visualizations, as well as improving the packaging so that the displays can each be seamlessly tessellated over the whole surface to form one larger display area.

6.3 New Applications

Aside from the applications that the DICE architecture is already capable of putting into practice, the addition of the displays enable other unique applications. Different forms of graphics rendering, model viewing, volumetric displays, and so forth were discussed, as well as the specific application of locally computed ray tracing. With such a recent project, the full array of potential applications has yet to be realized,

but there are surely more use cases that can be discovered. As these applications aren't possible without the DICE display framework, they serve as demonstrations of the merits of the system.

The ray tracing application in particular is one to get excited about because of the potential doors it opens in the world of graphics processing development. By scaling up the DICE-based GPU structure by adding more nodes and displays, intermediate steps between top of the line commercial product iterations could be achieved, bringing costs down and efficiency up. The whole idea of building the system to have just as much computational power as is needed, no more or no less, is now possible using this architecture.

6.4 Performance Projections

The basic DICE unit has been characterized and compared against various other computing chips and systems. According to performance projections, it would take roughly 3,000 DICE nodes to equal the performance of the Intel Core i7, while using 30% more power. [12] Based on these projections, it is worth exploring what additional computational and monetary costs are added on with the integration of the displays. Additionally, the basic performance of one singular DICE display node can be compared against top of the line GPU devices out on the market today, such as the NVIDIA V100.

First off, a discussion of monetary cost is required. Each DICE node costs roughly $3, and both the displays and the microcontrollers used for these tests cost approximately $20 each. Therefore, converting a top-layer node into a display node increases the cost more than tenfold. However, these specific devices were chosen for prototyping purposes and less expensive options could definitely pursued that could potentially fit the needs of the system even better.

In terms of power, a simple test was conducted where one node were placed on a build plate, which ran the particle simulation application drawing 60 mW of power. When connecting the display, the power consumption of the system increased by 130

mW up to 190 mW. Therefore, the power consumption of the system running a typical application can more than triple by adding in the display and its microcontroller. Again, by replacing these prototyping devices with something specifically designed for this system, this increase can be mitigated.

Coming back to the NVIDIA V100 comparison, however, shows how much work can continue to be done. Using the same math that compared DICE nodes to the Intel Core i7, we can determine that approximately 750,000 nodes would be required to match the performance of the NVIDIA V100, while using far more power. If 3,000 nodes didn't sound like it was pushing any limits, then 750,000 nodes is sure to sound absurd. Tack on the additional power consumption of the display layer and a potential decrease in practical computer power due to the need to slow the computations down so the displays can keep up, and it makes this system seem so inferior that its existence is called into question.

The thing to remember about all this, however, is that this system wasn't built to directly compete with these already existing products, but to provide an alternative architecture to potentially guide future developments in computing as a whole. These high-end computing devices are based on traditional non-locality based architectures. DICE is presented as an alternative that could potentially shape the next big wave of computing research.

It is worth noting, however, that there are other branches of this project being explored, one of them being the implementation of the DICE architecture using superconducting electronics. [5] According to current tests and projections, a custom-made "Super-DICE" node would utilize these superconducting advances to achieve computational and power consumption levels that are unheard of. Just 10 Super-DICE nodes could match the performance of the Intel Core i7, while using a fraction of a percent of the amount of power. Scaling up to the NVIDIA V100, it would take around 2,500 Super-DICE nodes to equal the performance, but again, only using a near-insignificant amount of power. These projections even show that if successfully implemented, it could lead to an improvement five orders of magnitude better than state-of-the-art super-computing systems. [12]

So again, while there is a lot of room for optimization, this system still gets the job done. Even though at its current state, these nodes would need to be scaled almost infinitely to match the levels of performance of modern computing structures, the fact that they work as well as they do is an improvement upon what has come before. There are various routes that the project can take in the future, but many, if not all, have the potential to achieve some very promising results and make progress in the fields of computing and graphics.

Chapter 7

Conclusion and Future Work

The work done in this thesis is hopefully another step forward not only for the DICE project as a whole, but for the whole field of computing and displays. The displays integrated into this system not only enhance the capabilities of the DICE nodes themselves, but enable other useful applications that wouldn't be possible otherwise. One of these applications is that of locally computed ray tracing, which could shape up to be a great advance in the GPU industry. There are some obvious limitations from the system as a whole, but there is quite a bit of work that can be done to overcome or minimize those limitations, making this system even more efficient and beneficial for those who put it into practice.

First, an obvious area for improvement that could go a long way in making the system easier to use is packaging. While this may seem superficial, it would certainly improve the user experience with programming the devices. The specialized header pin strut did its job for the testing conducted, but it would be even better to have everything designed so that the displays could mount and latch on top of each node just like the other nodes do to stack on top of each other. In order to do this, however, more testing needs to be conducted on displays to determine which ones would work best for the system. As different applications are developed, it will be more clear what the needs are in a display, especially in terms of resolution and color capabilities, which would further guide the decision for which display is best for the system. It was mentioned that the Feather M4 microcontroller was chosen to drive

the display because of its compatibility with the display, but also because it uses the same ATSAMD51 chip as the DICE nodes. While it might not be possible to drive the display using only a DICE node, a special display node could be designed to include the extra components necessary beyond the Meso-DICE components, and then could mount on top of a typical DICE node to fit in with the rest of the assembly process. It would not only be aesthetically pleasing, but also very practical, to find a design that helped create a tight tessellation between each display to create a seamless layer of tiled displays. All these improvements would be sure to not only make things look nicer superficially, but also make them work together better and make the development and testing processes easier. Concept art of how this idealized packaging could look is included in Figure 7-1.

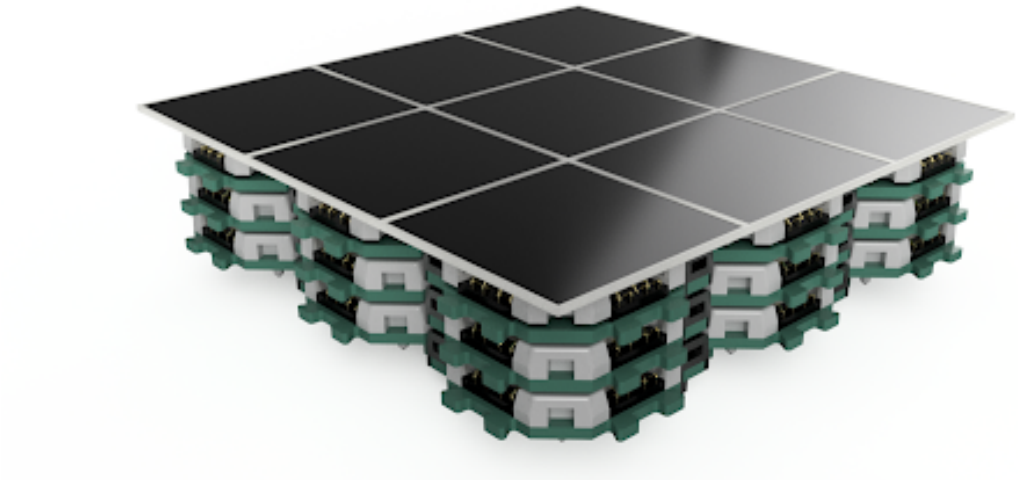

Figure 7-1: Concept rendering of an ideal Meso-DICE and integrated display packaging.

Next, work needs to be done on the transmission of data to the displays themselves. While the simple serial communication from the DICE nodes to the display's microcontroller seemed to do just fine for the testing conducted in this work, it is worth exploring how difficult yet beneficial it would be to have the same token passing scheme used between nodes put in place between the nodes and the displays. The handshaking and waiting functions would certainly help curb misread or unorganized data. The most important things that need to be maintained and improved, however,

is ease of use and speed. Whichever method can maximize both of those areas is the surefire winner to be adopted and maintained.

With those two areas improved, the system as a whole, especially in debugging and visualizations, will be lifted up. The visualizations will look much more appealing with displays seamlessly tiled and correctly placed above their respective node or node stack. By being able to just press in and power the displays in order to connect them and have them start displaying, the debugging process can be sped up, eliminating the current need for ensuring the header pin strut is in the correct location along with the jumper cables going to the correct pins. Soldered and secured connections are far more stable than a mess of jumper cables going between devices anyway, so all of this would sure to combine to lead to much more reliable performance.

The stability of the connections of the struts and nodes is something more to be explored in the overall DICE project. The connections held up extremely well for a given amount of time, but eventually started loosening up upon repetitive latching and unlatching. Many nodes lost one or more of their four heat-staked PLA posts holding the node together. Some struts integrated into the build plate also lost one side of the milled delrin latching piece. While not completely detrimental to operation, these breaks surely didn't make things more stable and secure.

One way that these structural issues could be mitigated is by having reprogramming capabilities implemented from node to node, and not just at the programming stand. By having a common backbone throughout the system, which partially exists in the build plate, the entire structure could be reprogrammed by sending the information to each node stack and having the nodes reprogram each other using the serial lines that connect them. This would make it so removing and replacing each component wouldn't be necessary, increasing the longevity of each of the components, not to mention the speed boost it would grant to development time.

As more time is dedicated to working through existing applications and improving them, the strengths of the system will become more apparent. The debugging capabilities should help any future developers work on different code and simulations within the system, as well as create appealing visualizations to make sense of the

applications they're working on. While current applications may be limited, each of them can be adapted to implement the displays to improve upon them in some way.

The particle simulation in particular has a lot to offer for this system, and vice versa. Once code is developed to get the physical DICE nodes interacting with one another and sharing particle information, the entire visualization could be quickly realized. From there, 3D implementations, as well as more complex particle interactions simulating stress fractures and other physical phenomena can be easily introduced.

For the ray tracing application, it is quite the bummer that it didn't make it past the conceptual stages, so the obvious next step is to bring it from the theoretical to the experimental realm. That will take quite a bit of work, but as it builds upon the foundation of the particle simulation, propagating photons should be a compelling application of the capabilities of the system. From there, more complex scenes and objects can be successfully rendered.

Another application mentioned in earlier chapters included that of cross-sectional view rendering. For this application, the DICE nodes could represent a volume with an object inside. Depending on the layer of nodes selected, a cross-section view could be rendered of the object at that layer and displayed on the top surface displays. Different layers would represent different depths in the display volume. A concept of how this could look is included in Figure 7-2.

Some other incredibly intriguing ideas are volumetric and holographic display applications. Specialized display hardware would most certainly be required to make the most of these applications. However, the parallel processing provided by the DICE nodes is perfect for the complex and robust computations that must be performed for things like computational holography. Similar to the ray tracing application, it requires computing the paths of the object beam and reference beam as it propagates through an environment, then to determine the diffraction pattern caused by the interference of the beams. It would be quite something to see a hologram computed and displayed on the same system.

While performance projections show that the project has a long way to go to be directly compared to modern computers, the specific direction that it takes could

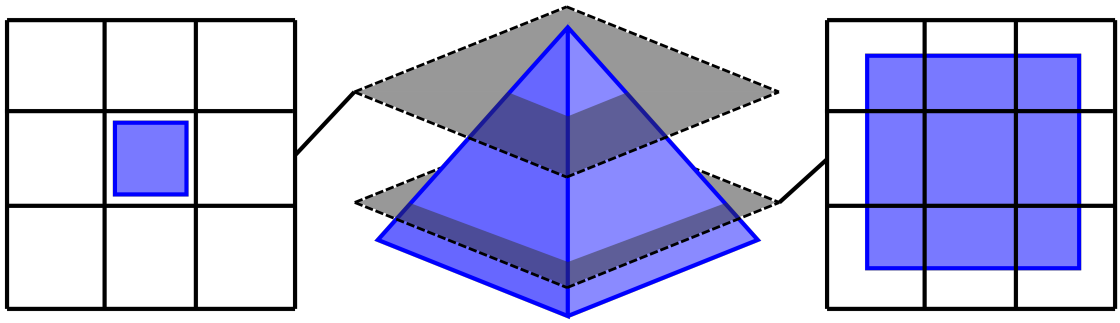

Figure 7-2: Concept of a cross-section view rendering application. The object in the middle would be included in the scene volume calculated by the display nodes. Depending on the layer selected, that layer of nodes would send data about the object from their layer to the displays, which would then render a cross-sectional view of the object. Different cross-sectional views can be seen represented in the display windows on the left and right.

potentially be among the first steps of a shift in computing architectures as a whole. Especially with the projections of the superconducting powered DICE nodes, the project could even be revolutionary. With the addition of the displays, it opens the doors to less conventional applications and implementations, but those should evolve with the project as a whole moving forward.

Again, it is anticipated that the contributions provided by this work are a great next step forward for all of the concepts explored. Much more research can and should be conducted to truly determine how impactful this system can be. As the system gets in the hands of more researchers and developers, hopefully its full potential can be realized. At the very least, this system is an interesting exploration of a unique computing and display architecture. However, if performance projections and other indicators are true, this could be another step towards a fundamental change in how the world looks at computing.

Bibliography

[1] Adafruit. "1.8" Color TFT LCD display with MicroSD Card Breakout - ST7735R." Accessed on: August 13, 2021. [Online]. Available: https://www.adafruit.com/product/358.

[2] Adafruit. "Adafruit Feather M4 Express - Featuring ATSAMD51 - ATSAMD51 Cortex M4." Accessed on: August 13, 2021. [Online]. Available: https://www.adafruit.com/product/3857.

[3] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering.* AK Peters/crc Press, 2019.

[4] J. Arvo. "Backward Ray Tracing." In *Developments in Ray Tracing, Computer Graphics, Proc. of ACM SIGGRAPH 86 Course Notes*, pp. 259-263, 1986.

[5] C. Blackburn. Concurrent master's thesis. Massachusetts Institute of Technology, School of Architecture and Planning, Program in Media Arts and Sciences, 2021.

[6] W. J. Butera. "Programming a Paintable Computer." PhD dissertation, Massachusetts Institute of Technology, 2002. [Online]. Available: http://cba.mit.edu/docs/theses/02.02.butera.pdf.

[7] P. Christensen et al. "RenderMan: An Advanced Path Tracing Architecture for Movie Rendering." *ACM Transactions on Graphics (TOG) 37*, no. 3, pp. 1-21, 2018.

[8] R. L. Cook, T. Porter, and L. Carpenter. "Distributed ray tracing." In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pp. 137-145, 1984.

[9] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. McKenzie. "Computational RAM: Implementing processors in memory." *IEEE Design & Test of Computers 16*, no. 1, pp. 32-41, 1999.

[10] A. J. Fewings and N. W. John. "Distributed Graphics Pipelines on the Grid." In *IEEE Distributed Systems Online*, vol. 8, no. 1, p. 1, Jan. 2007.

[11] Z. Fredin. Concurrent master's thesis. Massachusetts Institute of Technology, School of Architecture and Planning, Program in Media Arts and Sciences, 2021.

[12] Z. Fredin, J. Zemanek, C. Blackburn, E. Strand, A. Abdel-Rahman, P. Rowles, and N. Gershenfeld. "Discrete Integrated Circuit Electronics (DICE)." In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020.

[13] M. Gokhale, B. Holmes, and K. Iobst. "Processing in memory: The Terasys massively parallel PIM array." *Computer 28*, no. 4, pp. 23-31, 1995.

[14] G.R. Hoffman. "Who invented ray tracing?." *The Visual Computer, 6*, no. 3, pp. 120-124, 1990.

[15] G. Humphreys, I. Buck, M. Eldridge, and P. Hanrahan. "Distributed Rendering for Scalable Displays." In *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, p. 30. IEEE, 2000.

[16] G. Humphreys and P. Hanrahan. "A distributed graphics system for large tiled displays." In *Proceedings Visualization '99 (Cat. No.99CB37067)*, pp. 215-527. IEEE, 1999.

[17] D. R. Hutchings, J. Stasko, and M. Czerwinski. "Distributed Display Environments." In *CHI'05 Extended Abstracts on Human Factors in Computing Systems*, pp. 2117-2118, 2005.

[18] H.W. Jensen and P. Christensen. "High Quality Rendering using Ray Tracing and Photon Mapping." In *ACM SIGGRAPH 2007 courses*, pp. 1-es., 2007.

[19] K. Li et al. "Building and using a scalable display wall system." In *IEEE Computer Graphics Applications*, vol. 20, no. 4, pp. 29-37, July-Aug. 2000.

[20] N. Margolus. "CAM-8: a computer architecture based on cellular automata." 1993. [Online]. Available: https://people.csail.mit.edu/nhm/cam8.pdf.

[21] B. F. Mistree and J. A. Paradiso. "ChainMail: A Configurable Multimodal Lining to Enable Sensate Surfaces and Interactive Objects." In *Proceedings of the fourth international conference on Tangible, embedded, and embodied interaction*, pp. 65-72, 2010. [Online]. Available: https://resenv.media.mit.edu/pubs/papers/2010-01-fp245-mistree.pdf.

[22] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. "A Sorting Classification of Parallel Rendering." *IEEE computer graphics and applications 14*, no. 4, pp. 23-32, 1994.

[23] T. L. Sterling and H. P. Zima. "Gilgamesh: A Multithreaded Processor-In-Memory Architecture for Petaflops Computing." In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pp. 48. IEEE, 2002.

[24] E. Strand. "Inverse Methods for Design and Simulation with Particle Systems." Master's thesis, Massachusetts Institute of Technology, 2020. [Online]. Available: http://cba.mit.edu/docs/theses/20.09.strand.pdf.

[25] T. N. Theis and H.-S. P. Wong. "The End of Moore's Law: A New Beginning for Information Technology." In *Computing in Science & Engineering*, vol. 19, no. 2, pp. 41-50, Mar.-Apr. 2017.

[26] I. Wicaksono, J. Cherston, and J. A. Paradiso. "Electronic Textile Gaia: Ubiquitous Computational Substrates Across Geometric Scales." *IEEE Pervasive Computing*, 2021. [Online]. Available: https://resenv.media.mit.edu/pubs/papers/2021-Wicaksono-Cherston-EtextileGaia.pdf.