Lamancha - Ultimate Quip

Introduction

MISSION & STRATEGY

The ultimate mission of Lamancha is to make correctly implementing board games easy, quick, and ultimately cheaper from an engineering point of view. The strategy for achieving this is to build a generic client for devices (similar to a web browser) and then move all game logic to a single threaded server where “compute is durable” and easy enough such that high school hackers can understand the model.

VALUE PROPOSITION

The initial value proposition of this work is to achieve the mission for board games, but this is simply a mechanism to amuse the author (me, Jeff). At the core of this is a severe rethinking of how we build software in an interconnected world. We are seeing the cloud be a revolution for many areas, and the cloud is a powerful thing. However, we must ask what will be next? Will the cloud continue as it, or is there something that could severely disrupt it? The world is getting more complicated as to where data lives and who owns that data, so how can products get built quickly without regulatory concern?

The abstract claim is that this work will enable a new way to build products that connect people and help people live their lives in a predictable way. Software can become secure and private by default. Software should be as easy to build and operate as a spreadsheet.

This industry is in its infancy, and the real value of this arc is to get a glimpse of the future. Now, the author may be a crackpot, and the inverse value of this work is that the engineering discipline to operate this vision is a huge unknown. It may be too myopic in scope and toy-like, but many great things start as a toy.

REQUIREMENTS TO SUCCEED AT THE PRIMARY MISSION

- The rendering must be exceptionally efficient since board games can last hours. We expect that an average person with 50% battery available should be able to play a six hour game with a mature battery.
- Animation and expensive visual effects must be dependent on time and cancelled. That is, if an animation is desired and implemented then system configuration can override that animation and simply snap to the end state or skip frames to reduce battery.

NEXT STEPS

This document will outline the high level view of the entire stack, the components found within the stack, a specification for a programming language, and then conclude with examples that illustrate how the stack solves board games.

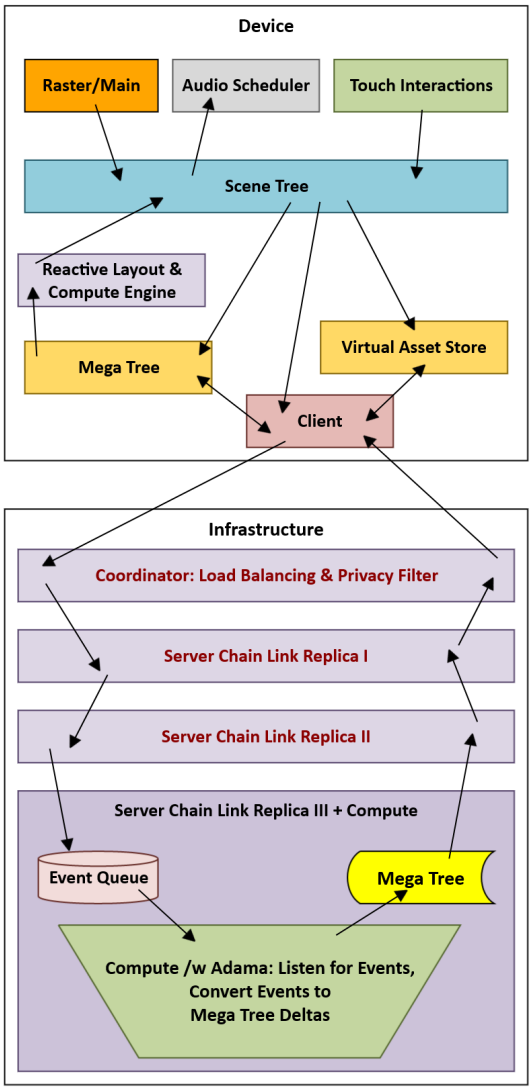
High Level Technical View

ARCHITECTURE AT 50,000 FOOT VIEW

This design is “full stack” as it encompasses everything from the device’s screen down to the server side language used to write the entire product. The image to the right illustrates the various components and their relationships between themselves. First, we will illustrate the broad

stroke roles of each component at a high level and how these components relate to each. Second, each component will have a section to call out important design details. Finally, we will conclude with examples of how all these components tie together to enable board games.

- **Raster Main** is responsible for drawing on the device's display. This produces the visual results of the product. At the same time, drawing will drive all other processes within the system. **Raster Main** is responsible for the game loop which will be non-traditional.
- **Scene Tree** is the classical "document model" that defines what the scene is composed of. For instance, it will contain the various items to draw and their positions. It also has audio ques embedded in it.
- The **Audio Scheduler** is driven by the rendering process such that the visual and audio effects are in alignment. **Raster Main** has the responsibility of walking the **Scene Tree** to update the display, and this walking process will schedule audio clips to play via the **Audio Scheduler**.
- **Touch Interactions** are the primary way on mobile devices for the user to interact with the product. These **Touch Interactions** will interface with the **Scene Tree** to convert the device's Touch Down, Touch Move, and Touch Up events into actualized events like "*Button 42 was touched which signals a roll of the dice*". The **Raster Main** component is also responsible for indexing active **Scene Tree** elements such that the touch coordinates can efficiently be converted into product level events.



At this point, the **Render Main**, **Scene Tree**, **Audio Scheduler**, and **Touch Interactions** feel similar to how a browser works without the JavaScript ecosystem. That is, they take a document, draw it on the screen, index it, and then provides mechanisms to convert finger interactions into useful behaviors (like change which document to render). The big question is how do we achieve interactivity with this system. This is where we introduce the **Reactive Layout & Compute Engine**.

The most direct analogy of **Reactive Layout & Compute Engine** is an Excel spreadsheet. Properties of the **Scene Tree**, like the coordinates of a image, are not always static values, but instead they may be expressions. These expressions resolve against both the **Mega Tree** and the **Virtual Asset Store**. The **Mega Tree** is the application state for the entire product. The **Mega Tree** is the core storage and where all state is kept. The **Virtual Asset Store** is effectively a blob store for images, audio, videos, fonts that can be referenced by the **Scene Tree**. As an additional bonus, video streams should be hooked up to the **Virtual Asset Store**.

In a way, the **Reactive Layout & Compute Engine** is a pure stateless function for converting the **Mega Tree** with assets pulled from the **Virtual Asset Store** into an image and audio events. Interactivity and animation are then achieved via

changes to the **Mega Tree** and **Virtual Asset Store**. Changes to the **Mega Tree** occur in four ways:

1. **Touch Interactions** fires an event that mutates the **Mega Tree.** It is worth noting that these mutations are limited in scope and will not be Turing complete. Later in the document, we will call out the balancing act of what the client is able to do locally without a server (see ⚓The Balancing Act; Dumb But Not Too Dumb Client)
2. **Render Main** detected an audio cue based on data in the **Mega Tree** and writes back that the audio is playing. The core reason for this write back to the tree is to signal that the audio is playing. Initially, this will serve as a way to have a falling edge to prevent audio from looping.
3. Temporal changes (or convergent) behaviors to animate scenes. Time is an input of the **Reactive Layout & Compute Engine** such that animation can be achieved by expressions. The goal here is invent a new computational form of relativity which illustrates why there can not be a universal clock.
4. Changes (i.e. tree diffs) arrive from the **Client**. This is a dominant way for updates to arrive as they will be coming over the network from a server. There is much to discuss with the **Client**, but a key idea is that tree will receive tree updates and then update the display.

Similarly, changes to the **Virtual Asset Store** depend entirely on the **Client** as well. At this point, we must contend with the role of the **Client.** The role of the **Client** is four fold:

1. Learn of **Mega Tree** updates from the server via the network. The server will emit a series of tree changes over time, and the **Client** will learn of them and dispatch them to the **Mega Tree**. Changes to the **Mega Tree** will then trigger rendering and audio plays.
2. Learn of new or updated assets for the **Virtual Asset Store.** These updates also arrive from the server via the network, and it is entirely possible for an ad-hoc video protocol to be born. As part of this, various asset types will also have a set of mutations which the server can apply to locally update an asset. For instance, an image asset could be updated by drawing a smaller image within an existing asset.
3. Durably propagate events generated from **Touch Interactions** interfacing with the **Scene Tree**. This is the primary method of the server learning of interactions, and these must be queued until the server acknowledges them.
4. Collapse and gossip local **Mega Tree** changes to the server. This is a secondary form of server learning of state changes, but only limited in-flight changes are ever sent to the server which will impose a latency penalty during high state change. However, while a change is waiting to propagate to server, changes can collapse into one and overwrite changes. This means that this stream of data can not be used for triggering actions. Instead, it can be used for optimization or analytics.

The **Client** will connect to the first server which has a variety of roles and thus has the opaque name of **Coordinator**. This is where things go into the madness section as the server stack uses chain replication. However, unlike chain replication as proposed in “Chain Replication for Supporting High Throughput and Availability” (http://www.cs.cornell.edu/home/rvr/papers/OSDI04.pdf); this chain replication will be bi-directional. The **Coordinator** server has five jobs.

1. Identify the replicas that will define the chain (i.e. replica host A, B, and C). This step is effectively to identity the chain of servers via some shard placement service which must provide a consistent and partition tolerant view of the data.
2. Connect to the first replica, **Server Chain Link Replica I**, in the chain.
3. Proxy all events from **Client** to first replica in the chain
    a. All state mutations into a special type of event. State mutations within the infrastructure do not collapse due to the relatively higher throughput between servers and the waste of trying to batch overtime.
4. Store a copy of the mega tree and keep it fresh.
    b. The first connect to the first replica will download the entire document
    c. Once connected, updates will come from the first replica
5. Evaluate privacy rules within the mega tree and filter the document and updates to the client

d. Privacy rules are embedded within the document so the document is a closed data source.
e. Example: how to prevent clients from hacking their device to see other player's private state.

Server Chain Link Replica I and **Server Chain Link Replica II** have relatively simple jobs during normal operations.

Server Chain Link Replica I will receive events from **Coordinator** and then store them and pass them to **Server Chain Link Replica II**. **Server Chain Link Replica II** will then produce data in the form of compete trees or updates, and **Server Chain Link Replica I** must store them and pass it along to the **Coordinator**. **Server Chain Link Replica I** is then a simply proxy that captures everything. **Server Chain Link Replica II** does exactly the same thing as **Server Chain Link Replica I**, and the primary reason for this durability. **Server Chain Link Replica II** then bridges from **Server Chain Link Replica I** to **Server Chain Link Replica III** which has a new role of **Compute.** This is where things get interesting, and we can make the observation *"Hey, all these replicas are just storing data!"*

Compute has one job, and that job is to convert events into state changes. State changes then come annotated with event deletions such that updates are atomic. This means that the **Compute** side must be entirely deterministic to handle a variety of failure modes, but it also means that **Compute** can move around on failure. The challenge of **Compute** then is an expression problem as this model potentially creates a variety of issues that can be mitigated by either discipline or a new programming language. The new programming language, **Adama**, will be specified to illustrate the discipline requirements.

SUMMARY OF THE SUMMARY; 100,000 FOOT VIEW

Components

Component: Scene Tree, Part I - Drawing

Reminder, the **Scene Tree** is responsible for the representation of what is visually and auditory being presented to the user via the device. For clarity, the **Scene Tree** in this document will represent 2D image construction and stereo audio mixing.

REPRESENTING THE TREE

The default representation of the **Scene Tree** begins in XML, and this is because XML makes representing hierarchical documents easy and straightforward (a bit ugly, but that is OK). Unfortunately, XML tends to be difficult to work with across platforms and is not efficient for devices to parse. Therefore, we will leverage XML for the very human developers and then introduce two alternative isomorphic representations.

The first alternative is JSON, and this representation is to make it easier for developers to transform and work with scenes in other tools and languages. JSON is easy to work with and reasonably efficient when compared to XML, and this opens up the potential for an ecosystem to develop to play with **Scene Trees**.

The second alternative is byte code, and this representation is for the end client to execute and render. An idealized implementation of the byte code would be WebAssembly because inventing a byte code format for this would be insane. This fundamentally means the translation to byte code requires converting the XML to a programming language, like rust, and then compiling that language to WebAssembly.

Since we expect humans to understand the **Scene Tree** via XML, the following sections will explain the basics using XML trees as examples and build up the elements from simple to complex. As elements are introduced, various design games will be laid out like "how does layout work" since this will illuminate elements.

XML: KICKING IT OFF WITH A FEW PRIMITIVE ELEMENTS AND THE KISS PRINCIPLE

Before we start laying out the foundation of the **Scene Tree**, we first layout the design aspiration that every element should be "simple", and here we define simple as being having one job to perform. For instance, let's consider the FillColor element.

```
<FillColor Color="red" />
```

This element has one job, and that job is to paint the screen's black surface with red. Now, there is more to life than painting black screens red. For instance, how would I shape that fill? This is where we introduce the **Scope** element.

```
<Scope Width="64" Height="64" Unit="px"><FillColor Color="red" /></Scope>
```

The **Scope** element has one job, and that job is to scope the size of the children to 64x64 pixels. This document will simply have a red box with dimensions of 64x64 pixels sitting in the top left. (TODO: picture)

The width and height attributes define the dimensions of the element, and the unit attribute defines the unit for those measure. As a further example of keeping things simple, we expect all attributes to be basic atomic types like numbers, strings, a massive color enumeration that every system in the universe understands, enumeration values, and a few more. At no point do we expect developers to know "micro-languages" for the XML like "64px" (well, beyond the massive color enumeration encoding).

With the ability to size children, we also need the ability to move children elements and this is where we must introduce the **Translate** element.

```
<Translate X="42" Y="13" Unit="px">
  <Scope Width="64" Height="64" Unit="px">
    <FillColor Color="red" />
  </Scope>
</Translate>
```

The **Translate** element has one job, and that is translate children by the specified measure. With this document, there is a red box with dimensions 64x64 pixels located at the coordinate (42, 13) measured in pixels from the top left corner. (TODO: picture).

So far, we have introduced three elements: **FillColor**, **Scope**, and **Translate** but there are two games being played. The first game is "how are elements sized?", and the second is "how are elements positioned?"

GAME: INHERITANCE BASED SIZING

An important design question is "how do we size elements?". The size of an element is actually an input from the containing parent's elements. At the root of the document, the size the parent tells children is the device's window. For this document, all devices will have a bounds of 600 pixels horizontally by 800 pixels vertically. Children will then accept the parent's size as a suggestion and then can use it. Consider this document:

```
<FillColor Color="blue" />
```

The size of the **FillColor** element will be (600, 800) as this what the device's size is. Now, consider this document:

```
<Scope Width="75" Height="50" Unit="percent">
  <FillColor Color="blue" />
</Scope>
```

The **Scope** element will receive the bounds (600, 800) from the device and then it will do its one and only job of converting that to *(600*75/100, 800*50/100)* = (450, 400) pixels for children. **FillColor** will then get (450, 400) and use that for the size and ultimately what to render.

This system is a stark contrast from other systems where size comes by sizing the children elements and up, and we want to make clear that the mode of sizing is available as well but will be detailed later on the the document. A core reason for this model is that it is exceptionally efficient in terms of execution and aligns with the implicit goal of a battery-efficient.

GAME: THE MATRIX STRACK

After sizing elements, we now must ask “how are elements positioned?” This question is answered with matrices! OK, we have the Translate element like:

```
<Translate X="42" Y="13" Unit="px">
  <FillColor Color="green" />
</Translate>
```

Behind the scenes, there is a matrix stack. The matrix stack initially has an identity 3x3 matrix pushed on its top, and the above translation corresponds to the the 3x3 homogeneous matrix:

```
[1, 0, 42,
 0, 1, 13,
 0, 0, 1]
```

which is represented as a 3x2 matrix where the bottom row is fixed at 0, 0, 1 as in:

```
[1, 0, 42,
 0, 1, 13]
```

This matrix is multiplied by the head of stack (i.e. the identity matrix) and then the result is pushed on top of the stack for other matrix operations. The stack is then used by children of **Translate**, and once the children are processed, the result is then popped off the stack.

This matrix allows many affine transformations to be encoded like rotation, scaling, reflection, shearing, and orthogonal projection.

XML: EXPANDING THE MATRIX STACK

Since the matrix stack is introduced, now let’s introduce the **Scale** and **Rotate** elements which also manipulate the matrix stack. **Scale** will simply enlarge or shrink or the children in question, and has two attributes.

```
<Scale X="2" Y="2">
  <Scope Width="32" Height="32" Unit="px">
    <FillColor Color="orange" />
  </Scope>
</Rotate>
```

This **Scale** element will double the size of the children, and this has side effect of turning the specified 32x32 orange box into a realized 64x64 orange box. Similarly, an X and Y value of 0.5 would halve the orange box’s size. Fortunately, the **Scale** element object is simple. **Rotate**, however, is intuitively simple but has a rough edge; **Rotate** will rotate the children by the given angle in the given angle measurement. For instance,

```
<Rotate Angle="45" Unit="degree">
  <Scope Width="32" Height="32" Unit="px">
    <FillColor Color="purple" />
  </Scope>
</Rotate>
```

will rotate the purple box by 45 degrees. **Rotate** is an interesting case study because there is now a good question of what is the size of the rotated children? The size of the children is 32x32 prior to rotation, and the rotation will naturally increase the bounds of that box. So, what do we do?

First, we have to recognize that this problem is related to a future section of automatic layout and how elements can be position. For instance, if elements are marshaled left to right, then contending with size is important such that elements don't overlap if desired. The tricky element at play here is that there is implicit hidden state; the size of the children is hidden and may not be available within this document. The language should provide some mechanism to aide in making that hidden state useful and provide a few options.

There is a tiny explosion of complexity here, and this is why we value keeping things simple to contain this explosions as they happen. We have several options. First, we could simply ignore the effect of rotation and **Keep** the children's bound as-is. Or, we could **Expand** the size such and implicitly induce a **Translation** to preserve a box model and fit the rotated item inside the box. Alternatively, we could **Shrink** the parent's bound and induce a **Scale** and **Translation** such that the children's rotation will fit within the provided box. We will call this decision as a **SizeEffect** which is enumeration with values **Keep**, **Shrink**, and **Expand** and then add a ***SizeEffect*** attribute to the Rotate element. Consider this document:

```
<Rotate Angle="45" Unit="degree" SizeEffect="expand">
  <Scope Width="32" Height="32" Unit="px">
    <FillColor Color="purple" />
  </Scope>
</Rotate>
```

The size of the **Rotate** element is around ~44x44 (TODO: sort this out this math) pixels, and the purple box will be centered within that 44x44 box such that the the corners of the rotated box touch the edges. It is worth noting that none of these options are battle-tested with designers, and **Keep** will most likely be the primary which is what browsers do with CSS.

Additionally, we must contend that hidden state of the children size may be useful to affect which point the element is rotated around. For instance, should the box rotate around the children's like how CSS rotate() works? or should the rotation happen on the left corner of the children? This is where we introduce the **OriginX** and **OriginY** attributes which default to 0.5 to represent the center and maintain a bit of semantic comparability with browsers behavior with CSS rotate(). For instance, consider the document.

```
<Rotate Angle="180" Unit="degree" OriginX="0.5" OriginY="1.0">
  <Scope Width="32" Height="32" Unit="px">
    <FillColor Color="purple" />
  </Scope>
</Rotate>
```

It will render the box rotated around the center bottom point which has the effect of turning the box upside down (which, in this example is still a purple box...). This begs an interesting question of how could the hidden state of the children's size affect the design of Translate, and the answer is to introduce a new unit called ChildSize such that the following document:

```
<Translate X="0" Y="-1" Unit="ChildSize">
  <Rotate Angle="45" Unit="degree" SizeEffect="keep" OriginX="0.5" OriginY="1.0">
    <Scope Width="32" Height="32" Unit="px">
      <FillColor Color="purple" />
    </Scope>
  </Rotate>
</Translate>
```

can effectively flip over a box and then render the box in the same position.

XML: SCOPING VERSUS DRAWING

Rotation introduces an interesting effect where the drawing of the children may exist outside of the size and relative position of a box. We also may have interesting documents like this:

```
<Scope Width="32" Height="32" Unit="px">
  <Scope Width="64" Height="64" Unit="px">
    <FillColor Color="pink" />
  </Scope>
</Scope>
```

This definitely feels strange, but it is valid and will render a pink box with dimensions 64x64 pixels. This is where we can introduce the **Clip** element.

```
<Scope Width="32" Height="32" Unit="px">
  <Clip>
    <Scope Width="64" Height="64" Unit="px">
      <FillColor Color="pink" />
    </Scope>
  </Clip>
</Scope>
```

This **Clip** element has one job, and that job is to clip the children's rendering to the parent's bound. The above example will clip the children's rendering according to the inherited bounds, and in this example a pink box will be rendered in the top left with size of 32x32 pixels. This has the effect of creating a portal into a potentially larger document. For instance, consider this document now:

```
<Scope Width="32" Height="32" Unit="px">
  <Clip>
    <Translate X="45" Y="75">
      ... Crazy Stuff
    </Translate>
  </Clip>
</Scope>
```

This starts to demonstrate how the concept of "panning" is an emergent property of the system if the X and Y values of the above **Translate** could change. However, this is starting to get ahead of us. Let's return to the fact that there is a negotiation of sorts between the parent sending bounds down and the children's ability to emit its size up based on the parent's bounds.

XML: NEGOTIATION BETWEEN THE BOUNDS OF THE PARENT AND CHILDREN

An interesting negotiation emerges between parent's bounds that flows to the element's children and what happens when the children have their own agenda with respect to their size, and this is where elements **Pad**, **Align**, **Zoom**, **Anchor**, and **AspectRatio** will come into play to help sort things out.

Pad has one job, and that is to reduce the bounds coming from the parent to the children and then increase the bounds the children emits. Consider this example:

```
<Pad Left="5" Right="5" Top="5" Bottom="5" Unit="px">
  <Scope Width="90" Height="90" Unit="px">
    <FillColor Color="teal" />
  </Scope>
</Pad>
```

This document will render a teal box 90x90 pixels with an invisible border of 5px all around. The visual effect of this document is equal to the teal box being translated by the vector (5,5), but the size the **Pad** element is 100x100 pixels. A similar document:

```
<Pad Left="10" Right="10" Top="10" Bottom="10" Unit="px">
  <FillColor Color="teal" />
</Pad>
```

Will render a teal box with bounds 590x790 offset by the vector (5, 50) yet the size of the **Pad** is 600x800.

Align has the job to align the edges between the parent and children as best as possible.

Zoom has the job to fit the children into the parent's bound.

Anchor has the job to constrain the distance between the parent's edges and the child edges.

AspectRatio has the job to adapt the parent's bounds to conform to the given aspect ratio.

XML: AUTOLAYOUT FOR EASY MATH

At this point, we have a great number of capabilities for explicitly putting elements in specific and exacting places, but how can we be a bit more lazy. For instance, how do I stitch things together? This is where we can introduce the concept of **AutoLayout** which is then fed by **AutoTranslate**. After, laying out a variety of elements is really a question

```
<AutoLayout Name="Foo" LayoutMode="LeftToRight" NextMode?="Center">
  <AutoTranslate Join="Foo">
    <Scope Width="90" Height="90" Unit="px">
      <FillColor Color="red" />
    </Scope>
  </AutoTranslate>
  <AutoTranslate Join="Foo">
    <Scope Width="90" Height="90" Unit="px">
      <FillColor Color="green" />
    </Scope>
  </AutoTranslate>
  <AutoTranslate Join="Foo">
    <Scope Width="90" Height="90" Unit="px">
      <FillColor Color="blue" />
    </Scope>
```

```
    </AutoTranslate>
  </AutoLayout>
```

XML: THIS IS TOO VERBOSE, MACROS AND FUNCTIONS FOR FUN AND PROFIT

XMacro is an element that is used to simplify the life of developers working with the **Scene Tree** manually.

Component: Mega Tree

With a basic sense of the Scene Tree and how it attaches to the Mega Tree, we will now lay out the specific semantics of the Mega Tree.

- Outline the types within the mega tree
  - numeric
  - string
  - bool
  - arrays of objects
  - objects
- Primary mechanism for inward data flow
- Tree stability and value references
  - every value can be held and is expected to be stable
- define the subscription mechanism and garbage collected subscription management

Component: Reactive Layout & Compute Engine

With the Mega Tree semantics, we now reveal how the Scene Tree can go from a static image to a dynamic engine. Every single numeric, string, and bool value within the Scene can be backed by an expression.

Component: Scene Tree, Part II - Multiplicity & Decisions

Ideas that require pulling state

- ForEach
- Repeat
- Enter
- If
- IfNot

Post Sizing

Component: Raster Main

With **Mega Tree** being the source of all data flow, we can reason that if the Mega Tree does not change then the scene does not change. This allows us to change the game loop to a form that becomes exceptionally battery efficient, and we can further raise the stakes that animation and thrilling experiences can be simplified to further increase battery efficiency.

A NEW GAME LOOP

A traditional game loop will periodically poll events from the network and input devices, and then dispatch those events to update the items within the scene. That is, below is a traditional game loop:

```
while (gameIsAlive) {
  // are there any updates from input devices or network
  var events = pollEvents();
  // if so, great! dispatch then
  events.forEach((event) => event.dispatch(sg));
  // have the scene update
  sg.update();
  // draw the scene
  sg.render(display);
  // sleep for a bit before re-polling
  sleep(20 /* ms -> 50 fps */);
}
```

This works generally well and enables all forms of animation and a bunch of implicit work to happen. However, for achieving the goal of battery efficiency we must wield the fact that the scene is static if no updates to the Mega Tree have occurred. This means that we are allowed to simply wait for events to happen. The new game loop in this world is as follows:

```
while (gameIsAlive) {
  // sleep until events are available
  var events = await genEvents();
  // dispatch events
  events.forEach((event) => event.dispatch(sg));
  // did any of the events invalid anything draw
  if (sg.display_dirty) {
    // re-render the scene
    sg.render(display);
  }
}
```

This has many advantages with respect to CPU and GPU resources, but we must be honest that this will cost a great deal of memory. This is why memory pressure is considered a key metric in the design of how to subscribe to changes on the **Mega Tree**, and a reason the subscription model is imprecise because the goal is not to achieve perfection but a significant reduction in wasted resources by balancing CPU versus memory.

Component: Scene Tree, Part III - Time & Animation

Animation is a nice touch, but it is expensive. We hold that animation should be entirely

XML: SKIPPABLE ANIMATION

- LocalTime
- Converge

ANIMATION INFLUENCES THE GAME LOOP

This influences the game loop

```
// we prevent the infinite waiting on events
int nextStepMs = 0;
while (gameIsAlive) {
  // sleep until events are available
  var events = await genEvents(nextStepMs);
  // dispatch events
  events.forEach((event) => event.dispatch(sg));
  // did any of the events invalid anything draw
  if (sg.display_dirty) {
    // re-render the scene
    nextStepMs = sg.render(display).timeUntilNextRender;
  } else {
    // the scene has nothing scheduled, come back in a very long time
    nextStepMs = 60000;
  }
}
```

Component: Touch Interactions

While many apps benefit from multi-touch interactions, we bias around a single finger design since single finger designs are accessible by more people. Furthermore, we bias towards simply touching items rather than any gestures. This is not to say that multi-touch gestures will be forever removed from the platform, but we recognize the inherit challenges of using them effectively.

Effectively, we will concern ourselves with the first finger touching the screen, tracking that finger, then when that finger is off screen to evaluate events. The tricky bit is that we must have an agreement between where the finger started and where the finger ended to convert to an event.

Beyond simple taps, we will also consider what it means to drag the finger.

WHAT HAPPENS DURING RENDERING

Some elements will have the ability to register is box that can be tapped or dragged. There is an interesting question of whether or not these boxes are indexed per a quad tree, or if a list of tappable items are used. We can reasonable start with a list of items that can be tapped, and then index as needed. This will be a more battery effciient approach since rendering will happen more often than tapping, and we should bias

Component: Scene Tree, Part IV - Interactivity

We will consider four primary mechanism for local interaction and the finger. The first is Pan which allows the device to pan (or scroll) the screen on their device. This provides low latency smooth visual effects. The second is the ability to

- Pan
- OnTap / Interact (think about it)
- LocalTranslate
    - It should be noted that Dragg\

Component: Audio Scheduler

The Audio Scheduler is a simplistic view of audio and primarily provides a simple API to embed audio in the Scene Tree such that audio can be played in sync with the rendering of the scene. It's design is primarily to provide two forms of audio: background music and playing sound effects in response to data changes.

Component: Scene Tree, Part V - Audio

Since my audio experience is limited, I intend to focus on two forms of audio: Simple background music via Music, and then the ability to trigger a sound effect via Sound

- Music
- Sound

```
<Music source="file.mp3" link="variable" fadeoutms="4000" />

<Sound source="file.mp3" link="variable" />
```

Component: Virtual Asset Store

- images
- fonts

Component: Scene Tree, Part VI - Images & Text

XML: MORE TO LIFE THAN FILLING COLORS

FillColor is really a demo element that is useful for not introducing too much noise. The core elements for displaying real things are: Image, NinePatchImage, SubImage, TileImage, and Text

- Image
- TileImage
- NinePatchImage
- SubImage
- TileImage

XML: THE IMPORTANCE OF WORDS AND WRITING

- Text
- Fonts

Component: Client

- What clients gives to server
- What server gives to client

Protocol: Client to Server

- client → server
    - TreeMerge
    - Events
- server → client
    - TreeMergeAndEventDelete

Component: Coordinator

- story around privacy

PRIVACY FILTERS

- embedding privacy within a tree

SHARD MANAGEMENT

- finding three replicas

FAILURE MODES

- what happens when first replica fails

Component: Server Chain Link Replica

FAILURE MODES

- What happens when prior node fails
- What

Protocol: Server to Server

- client → server
    - WriteEvent
- server → client
    - AckEvent
    - MergeTreeAndDeleteEvents

Component: Compute

With the client able to bring to life the data stored durability, we have to outline the mechanics of how to compute and build product. Previously, we roughly outlined that **Compute** has the job of converting event(s) into state changes. Intuitively, this makes sense, but we need to outline the needed language requirements such that the conversion is reliable and predictable such that failure modes do not ruin our experiences.

CHALLENGES OF THIS MODEL

THE QUEUE CAN'T BE INFINITE

ASYNC/AWAIT ARE VITAL DESIGN ELEMENTS

DATABASE TECHNIQUES

Language: Adama

Adama is the programming language that powers the compute engine. **Adama** is a direct descendant of C except it lacks a heap, encourages global state (for a good reason), and then has a bit of fun with the typing system. **Adama** is highly opinionated.

Core Language

IT ALL STARTS WITH ATOMIC TYPES

The **Adama** languages has the common types of:

- boolean: bool
- integers: int{8, 16, 32}
- unsigned integers: uint{8, 16, 32}
- floating point: float, double

Adama also has common algebraic data structures like

- vector<T>
- set<T>
- map<D, R>
- pair<A, B>

Contrasting to other languages, these types are built into the language. Notably absent are strings, and strings are simply vector<uint8> which has the type alias of string.

STRUCTURES AND SIGNATURES

With the basic types and data container types, we can then combine them into structures

```
struct YourStruct {
  bool enabled;
  int32 cost;
  string name;
  map<string, string> properties;
}
```

Adama also introduces the concept of a structure signature.

```
signature HasEnabled {
  bool enabled;
}
```

Signatures provide an implicit and automatic way of writing code against a specification rather than an specific instance. For instance: an algorithm written against HasEnabled can accept YourStruct implicitly without any work. This is a type of duck typing.

SPECIAL CONSTANTS

- true / false
- pi, euler; these are built-in to the language out of respect
- empty is a stand-in for empty vector, empty set, empty map; it requires type inference for correct use and there are explicit forms
  - empty<map>
  - empty<vector>
  - empty<set>

UNARY OPERATIONS

- ! E
- - E
- ++ E / E ++
- -- E / E --

BINARY OPERATIONS

- +, -
- *, /, %
- <, <=, ==, >=, >, !=
- and, or, xor
- bitand, bitor
- ...
- E1 in E2

THE CORE CONTROL STRUCTURE

Adama supports traditional C control flow constructs like if/else.

```
if (true) {
  true_branch();
} else {
  false_branch();
}
```

or just a basic if statement

```
if (false) {
  // dead code
}
```

And while loops:

```
while (true) {
  statements();
}
```

And basic for loops:

```
for (int8 k = 0; k < 32; k++) {
  statements();
}
```

and do/while for kicks and giggles

```
do {
  // execute at least once
} while (false);
```

and a familiar yet very different switch statements

```
int k = ...;
switch (k) {
  case 4:
    break;
  case < 4:
    break;
  case > 10:
    break;
}
```

This warrants a discussion. The type of K will induce a variety of different operators. If the type of k is numeric, then specific values or ranges can be used. This enables greatly flexibility. If the type of k is a structure, then we also enable a limited form of structural matching

```
struct S {
  bool enabled;
  int k;
}

S s = ...;

switch (s)
  case .enabled && .k < 4:
    break;
  case .enabled && .k >= 5:
    break;
  case !.enabled:
    break;
```

Along with a few modern flow constructs like

- foreach (V in E) S
    - if E is a map<D, R>, then V will have the value pair<D, R>

    - if E is a set<T>, then V will have the type of T
    - if E is a set or vector<T>, V will have the type of pair<uint32, T>
- foreach (V of E) S
    - if E is a map<D, R>, then V will value the value of R
    - if E is a set<T> or vector<T>, then V will have the type of T

THE LANGUAGE OF DOING STUFF

Adama enables the writing of functions, and here we are going to be exceptional precise in that the function is actually a side-effect free function like; this is to mirror their definition in mathematics. These functions can only read the inputs and produce a single output.

```
function algo(int8 x) -> int16 {
  return x * x;
}
```

Now, this is very... limiting, and the way out of it is to write procedures which can then manipulate the inputs and return either no type (i.e. void return type) or a single type.

```
struct S {
  bool enabled;
  int8 k;
}

procedure initialize(S s) {
  s.enabled = true;
  s.k = 0;
}
```

NOTES ABOUT THE LANGUAGE, SO FAR

The language is a love child from VBA and C. The VBA inspiration is the bifurcation between functions and procedures, and this will be more clear in the future. A key element of this language is the memory model. There is no intrinsic malloc or new behavior, and all memory is derived from the containers.

WHAT ABOUT OBJECTS?

Fuck objects, they suck. Seriously, object orientated programming was mostly a mistake.

OK, HOW DO I ALLOCATE STRUCTS?

You don't

Data Container Language

The basic containers of vector, set, and map are straightforward enough. However, we also have table<T> which opens up a bunch of interesting stuff. So, let's define table

```
struct YourRow {
  string name;
  int8 age;
}
```

```
table<YourRow> table;
```

At this point, yourTable is going to behave exactly like vector<YourRow>. The reason for this is that it is a vector, and the key difference is tables provide indexing and constraints. For instance, we can extend the table such that the elements in the table are indexed by name such as

```
table<YourRow> table : name indexed;
```

Now, how then is this indexing leveraged? Well, simple; Adama has a built-in and somewhat limited SQL in it.

```
select name from table where name='Jeff';
```

this will have a type of iterator<string> which is a generator.

Built-in Giant State Machine

So, in a traditional language, the state backing an application represents the process. This is where **Adama** takes a very different approach. Adama treats all the backing state as a document, and the run-time then manages series of documents.

```

```

Run-time

Board Game Examples

A GENERIC LOBBY

CARDS AGAINST HUMANITY

DOMINION

BATTLE STAR GALACTICA