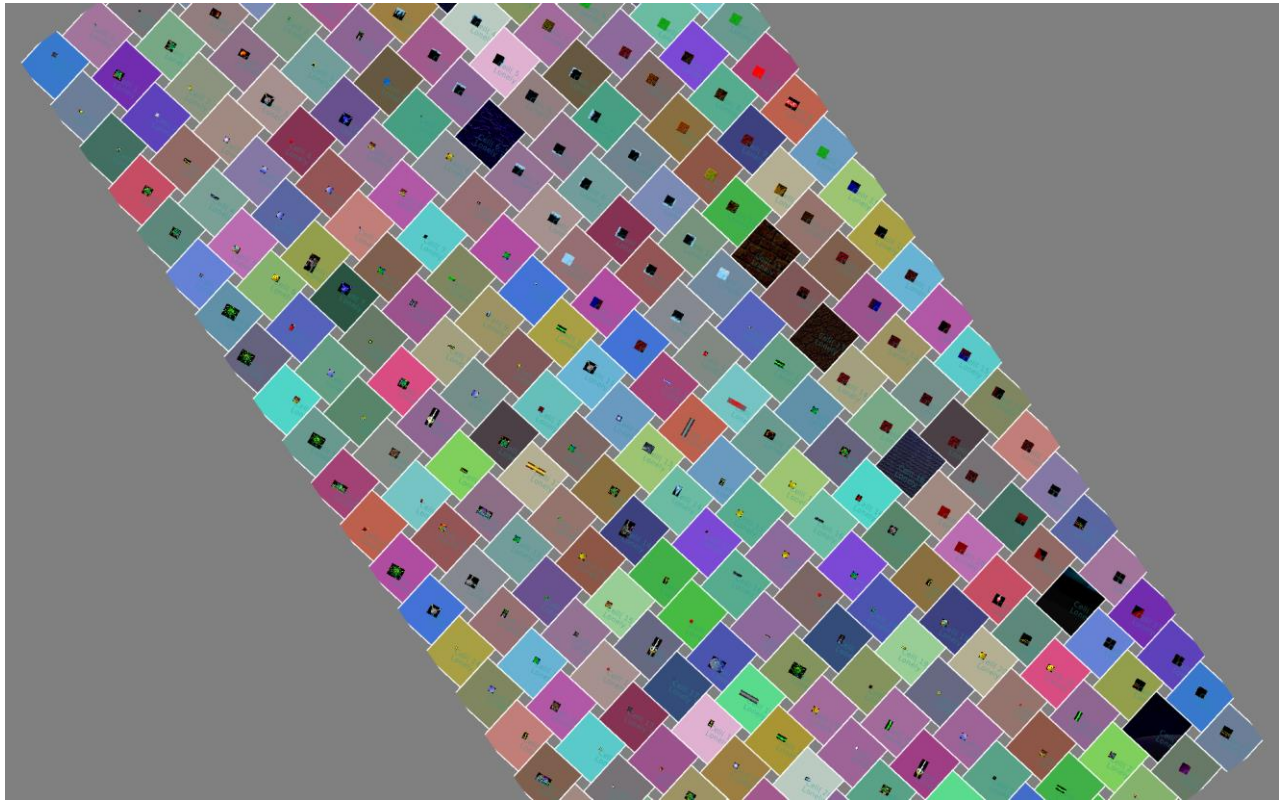
Fast UI Draw
UI Rendering for GPU's only

Kevin Rogovin

Benchmark: painter-cells

A. Draws a table of cells, each cell consists of
   1. a background,
   2. a rotating and moving image
   3. a rotating and moving line of text
B. In addition, strokes boundary (with anti-aliasing) between cells
C. Modes
   1. Can set table to rotate as a whole
   2. Can set cells to rotate separately
   3. Options on zooming, what to draw, etc.. can be changed while running (or at startup) too
D. Ported to Qt, Cairo and SKIA
   A. Checkout branch with_ports_of_painter_cells

Picture of painter-cells demo



Performance Numbers on ports of painter-cells demo to different libraries running 25x12 (600) cells. Normalized to Cairo CPU backend.

Canvas API	Performance
Cairo CPU	1.0
Cairo GL	0.27
Cairo Xlib	1.60
Qt Raster	1.01
Qt GL	0.76
Qt Native	0.29
SKIA GL	1.91
Fast UI Draw	**9.33**

Performance Numbers on ports of painter-cells demo to different libraries running 50x25 (1250) cells. Normalized to Cairo CPU backend.

Canvas API	Performance
Cairo CPU	1.0
Cairo GL	0.15
Cairo Xlib	1.25
Qt Raster	1.50
Qt GL	0.45
Qt Native	0.14
SKIA GL	1.36
Fast UI Draw	**11.11**

CONCLUSION:

The more complex the scene the higher FastUIDraw performs against other renderers.

REASON:

Fast UI Draw aims to leverage GPU's to render UI's

NOT SIMPLE! GPU's are for throughput and GPU's want bigger jobs with lots of the same work repeated. In contrast, CPU's are better able to handle changing work load/state.

GPU and CPU are very different beasts

CPU	GPU
No fixed function units.	Fixed functions units • Samplers • Rasterizers • Triangle setup and clipping
Very few rendering threads, each thread narrow. A. Switching threads within a (virtual) core is very expensive operation (Hyper-threading gives 2 virtual cores per physical core). B. Even with SSE, thread is only 4-wide or 8-wide A. SSE: 4-wide B. AVX: 8-wide	Many times more threads than cores, each thread quite wide. A. GPU core has register space for many threads; switching threads very cheap. B. GPU threads are quite wide. C. Example: Intel HD 4000 (IVB) has 16 EU's each EU has 7 threads. Thread width is 8 or 16 wide (depending on shader). D. GPU's hide latency with threads (and caching).

GPU has fixed function parts

- Fixed function parts have performance and power advantage over programmable parts
- For GPU's we have
  - Samplers
  - Rasterizer
  - Triangle Setup
  - Triangle Clipping
  - Depth buffer (with additional tricks usually to enhance performance)

Should leverage these as much as possible!

Goal: Minimize State Thrashing

If there is not enough work between state changes, a GPU work efficiency suffers.

Examples of state changes:

- Changing texture source
- Changing buffer sources or format
- Changing shader
- Changing depth/stencil test/operation
- Changing values of uniforms in shaders

Challenge for UI renderers to keep same API state even when changing what, how and where to draw

Fast UI Draw aims to have a single pipeline state

- Standard Canvas Features:
  - 3x3 transformation
  - Brush (image, gradients, brush transformation)
  - Stroking paths (with and without anti-aliasing)
  - Filling paths
  - clipIn
  - clipOut
  - Porter-Duff blend modes

- We also have features that are beyond canvas:
  - Custom shaders support (for example, non-linear distortions without needing to resort to render to texture first)
  - Stroke width can also be specified in pixels, even for non-orthogonal transformations

Blend Modes

1. I say aims to have. Currently, FastUIDraw needs 3 pipeline states for the GL backend

A. States are for supporting the 12 Porter-Duff blend modes (support all 12 Porter-Duff blend modes with just 3 different pipeline states). The only differences of the state is the blend mode.

B. Later, when support for W3C compositing modes are added, for some hardware, we can reduce the number of states to just ONE.

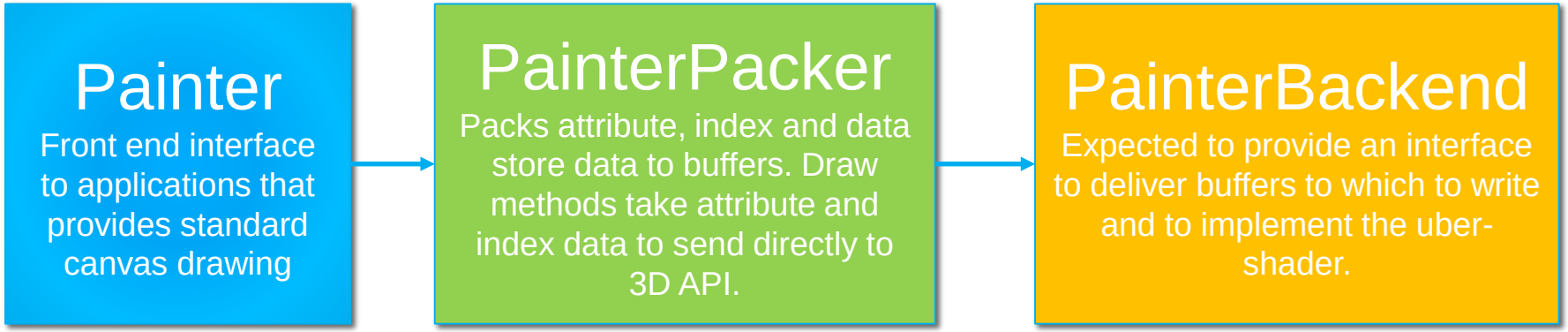
API Trace of Fast UI Draw rendering a frame is boring. Contents of an ENTIRE frame are:

1. Map Buffer, Unmap buffer
   A. Repeat a few times if lots to draw
2. Set GL API State (bind textures, use program, etc.)
3. Repeat a few times (same number as in 1A)
   A. Bind VAO, bind TBO, then repeat a few times:
      1. Set Blend State (only happens for changed involving half of the Porter-Duff blend modes)
      2. glMulitDrawElements (or glDrawElements)

Fast UI Draw uses an Uber-Shader and Data Store

1. Painter draw calls are always just: add data to mapped buffers
   a. Essentially 3 buffers: attribute buffer, index buffer and data store buffer
      - Data store buffer stores values that are shared between triangles (such as transformation matrix, brush properties)
      - Attributes and indices values are copied from the “what” (aka PainterAttributeData) directly.
2. When Painter draws are done, then 3D API draw calls are issued

High Level Architecture



A backend also includes implementing storage classes to hold the image, glyph and other data that the backend's implementation of PainterBackend uses.

Data Store Buffer

Data store buffer is a buffer which FastUIDraw packs data that a shader will unpack. Each call to a draw method of PainterPacker packs a header onto the data store buffer. The header contains what sub-shaders to use and offsets for various data:

Unnormalized depth value
Brush, Blend and Item sub-shader ID's
Offset for 3x3 transformation matrix
Offset for 4 clipping equations
Offset for brush data and item shader data

How data is packed is tightly specified so that creating a backend for other 3D API's (for example Vulkan) is possible.

FastUIDraw exposes an interface so that data already packed onto the data store buffer can be reused by an application.

Goal: Minimize re-computation of Data

- FastUIDraw distinguishes strongly between “what” and “how” to draw.
  1. The “what” to draw is computed once to be reused repeated
  2. The “how” to draw is small amounts of data fed to shaders. The how to draw includes:
     a. Brush (which image, which color stop sequence, gradient location, etc.)
     b. Position transformation
     c. Clipping
     d. Stroking parameters (stroke width, miter limit, dash pattern)
  3. Painter interface is organized so that it is natural to reuse data of “what” to draw.
  4. The “how” to draw is data that lands on the data store buffer

Generate Data to Minimize Work to Draw Repeatedly

Example: path stroking data

- Data created from a path for stroking is so that the same data can be directly copied to attribute and index buffers on changing stroke style
  - Stroking width
  - Miter Limit
  - Dash pattern (Not yet completed!)

Example: path fill data

- Paths are triangulated and broken into components keyed by fill rule
- Allows to have arbitrary fill rules when filling paths (or clipping by filled paths) and CPU usage is minimal after first time path is used for filling.

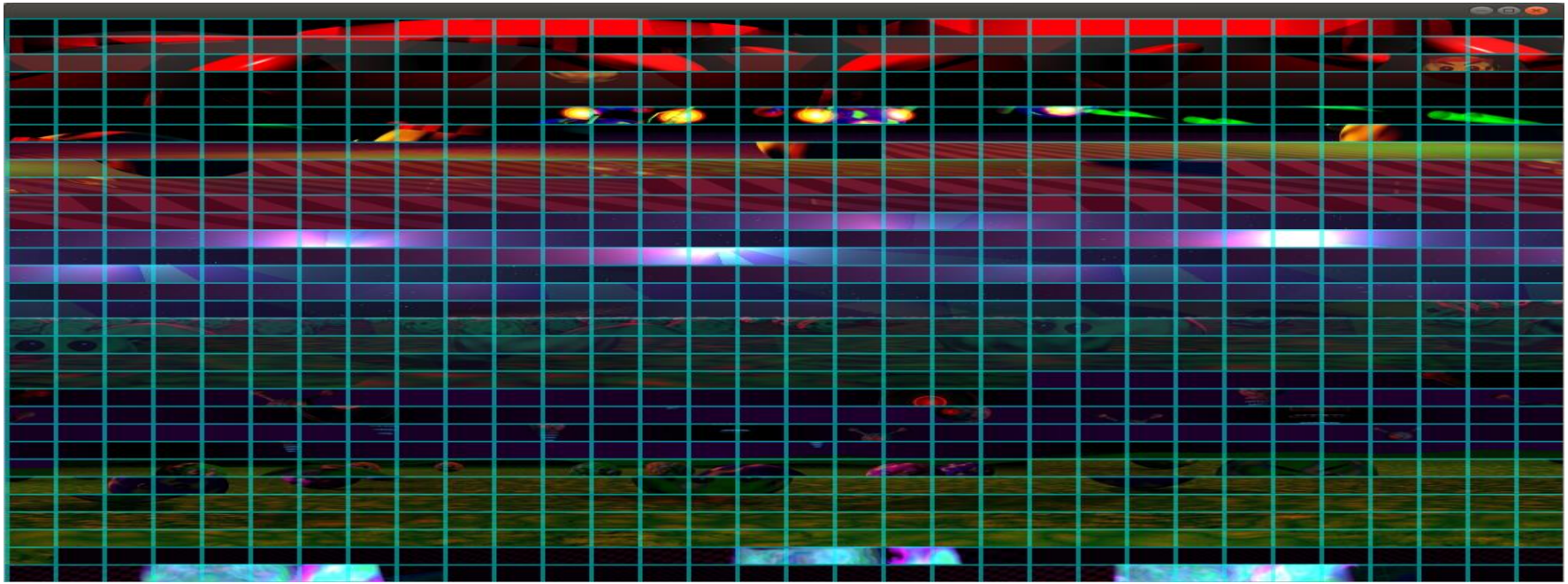
Issue: Many images in a UI scene

1. The idea of an image in a 3D API is embodied by the idea of a texture
2. Changing what texture (or textures) to use changes a portion of the pipeline state (namely the binding tables)
3. UI scenes contain many images of various sizes

Solution: Image Atlas for Padded Tiles

1. Images are broken into tiles of same size(size of tile is configurable but constant for lifetime of program)
2. Deallocating and allocating images is now trivial since we just need to know if enough tiles are free.
3. An image has also an index tile. The index tile gives the locations into the color tile texture to use.
4. The index tile values can also instead refer to an index tile instead. This allows us to support huge images (beyond hardware max texture size) without issue (subject to memory availability).
5. Giving padding to the color tiles allows us to use the GPU sampler to perform bilinear filtering and to also leverage it to accelerate bicubic filtering.

Picture of color atlas (taken from image-test) (Images packed are from Blob Wars' gfx/cutscenes)



Issue: Clipping

1. Canvas rendering means we can clipIn or clipOut against paths.
2. We aim to leverage GPU to handle the clipping without requiring the CPU to “compute” or track the clipping region.

Solution: HW Accelerated Clipping

FastUIDraw does NOT track the contents of the clipping region. FastUIDraw relies entirely on the GPU to implement clipping efficiently.

- The Depth buffer is used to implement clipOut:
  1. Path by which to clipOut is set to draw before that which it occludes
  2. The depth value (stored as a single value in data store buffer) is selected after the items it occludes are placed on the attribute, index and data store buffers
- clipIn by a rectangle is implemented by HW clip planes (in GL, gl_ClipDistance)
  - If transformation between last clipIn does NOT preserve the x-axis and y-axis, then complement of previous clipIn rectangle is 4 quads by which we clipOut. These quads are clipped to the new clipping rectangle to minimize how many depth pixels are touched.
- clipIn by filled path is just:
  1. clipIn by rectangle of bounding box of filled path
  2. clipOut by complement fill rule of path

Current State

A. Most Canvas features ready:
   1. 3x3 transformation
   2. Brush (image, linear gradient, radial gradient, brush transformation)
   3. Stroking with or without anti-aliasing
   4. Filling with arbitrary fill rule
   5. clipIn by Path or rectangle
   6. clipOut by Path
   7. GPU glyph rendering

B. What is not yet done
   1. Glyph renderer has issues with some glyphs, need to implement “more expensive” glyph renderer for these glyphs
   2. Anti-aliasing for filling paths
   3. Dashed stroking (in progress)
   4. Productization (!)

Try it out! Give me Feedback!

A. Clone from https://github.com/01org/fastuidraw
B. Read the docs (make docs)
C. Play with the demos
D. You can even “make install”
E. Send feedback to me
    A. through GitHub page (issues, etc.)
    B. or my email: kevin.rogovin@intel.com

Q & A