Renderscript

Lecture 10

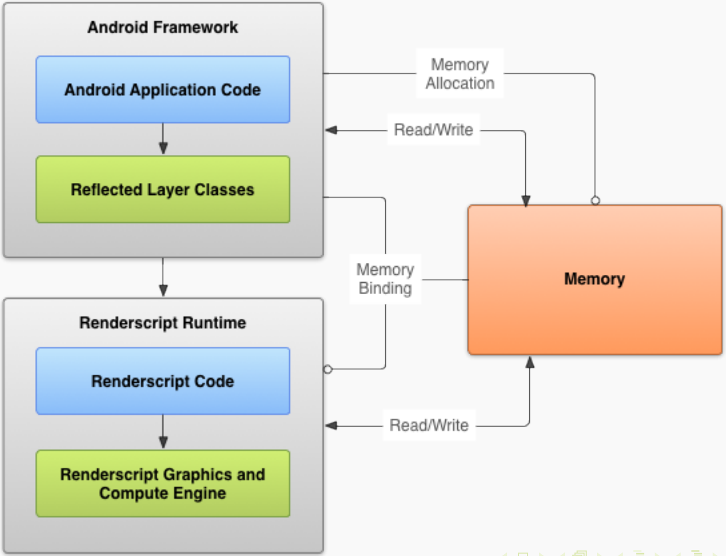
Android Native Development Kit

6 May 2014

- NDK - perform fast rendering and data-processing
- Lack of portability
  - Native lib that runs on ARM won't run on x86
- Lack of performance
  - Hard to identify (CPU/GPU/DSP) cores and run on them
  - Deal with synchronization
- Lack of usability
  - JNI calls may introduce bugs
- RenderScript developed to overcome these problems

- Native app speed with SDK app portability
- No JNI needed
- Language based on C99 - modern dialect of C
- Pair of compilers and runtime
- Graphics engine for fast 2D/3D rendering
  - Deprecated from Android 4.1
  - Developers prefer OpenGL
- Compute engine for fast data processing
- Running threads on CPU/GPU/DSP cores
  - Compute engine only on CPU cores

- Android apps running in the Dalvik VM
- Graphics/compute scripts run in the RenderScript runtime
- Communication between them through instances of reflected layer classes
    - Classes are wrappers around the scripts
    - Generated using the Android build tools
    - Eliminate the need of JNI
- Memory management
    - App allocated memory
    - Memory bound to the RenderScript runtime - memory accessed from the script
    - Script may have additional fields to store data


Android Framework
Android Application Code
Reflected Layer Classes
Memory Allocation
Read/Write
Memory Binding
Memory
Renderscript Runtime
Renderscript Code
Renderscript Graphics and Compute Engine
Read/Write

- Asynchronous call to RenderScript runtime to start script
  - Through the reflected layer classes
- Low-Level Virtual Machine (LLVM) front-end compiler
  - When building the apk
  - Compiles code into device-independent bytecode stored in the apk
  - The reflected layer class is created
- LLVM back-end compiler on the device
  - App is launched
  - Compiles bytecode into device-specific code
  - Caches the code on the device
- Achieves portability

- Java code and `.rs` file (compute script)
- API in `android.renderscript` package
- Class RenderScript
  - Defines context
  - Static `RenderScript.create()` returns class instance
- Class `Allocation`
  - Input and output allocations
  - Sending data to, receiving data from script
- Reflected layer class
  - Name: `ScriptC_` + script name
  - If script = computation.rs => class = ScriptC_computation
- `.rs` script
  - Placed in `src/`
  - Contains kernels, functions and variables

```
#pragma version(1)
#pragma rs java_package_name(com.example.hellocompute)

//multipliers to convert a RGB colors to black and white
const static float3 gMonoMult = {0.299f, 0.587f, 0.114f};

uchar4 __attribute__((kernel)) convert(uchar4 in){
  //unpack a color to a float4
  float4 f4 = rsUnpackColor8888(in);
  //take the dot product of the color and the multiplier
  float3 mono = dot(f4.rgb, gMonoMult);
  //repack the float to a color
  return rsPackColorTo8888(mono);
}
```

```
private void useScript() {
  RenderScript mRS = RenderScript.create(this);
  Allocation input = Allocation.createFromBitmap(mRS,
                                        inputBitmap);
  Allocation output = Allocation.createFromBitmap(mRS,
                                        outputBitmap);
  ScriptC_mono script = new ScriptC_mono(mRS);
  script.forEach_convert(input, output);
  output.copyTo(outputBitmap);
}
```

- `#pragma` to specify RenderScript version
  - `#pragma version(1)`
  - Version 1 is the only one available
- `#pragma` to specify Java package name
  - `#pragma rs java_package_name(com.example.app)`
- Global variables
  - We may set values from Java - used for parameter passing
- Invokable functions
  - Single-threaded function
  - Called from Java with arbitrary arguments
  - For initial setup or serial computations
- Optional `init()` function
  - Special type of invokable function
  - Runs when the script is first instantiated

- Static global variables and functions
  - Cannot be set/called from Java
- Compute kernels
  - Parallel functions
  - Executed for every `Element` within an `Allocation`
  - `__attribute__((kernel))` -> RenderScript kernel
  - `in` -> one Element from the input Allocation
  - Returned value put in one Element from the output Allocation
  - Multiple input/output -> declared global with `rs_allocation`
- Default root function
  - A special kernel function used in older versions
  - Returns void
  - `__attribute__((kernel))` not needed

Using RenderScript from Java Code

- Create RenderScript context
  - Using `create()`
  - May take a long time
- Create at least one `Allocation`
  - Provides storage for a fixed amount of data
  - Input and output for the kernels
  - Created with `createTyped(RenderScript, Type)` or `createFromBitmap(RenderScript, Bitmap)`
- Create script
  - User-defined kernels
    - Instantiate `ScriptC_filename` class
  - ScriptIntrinsic - built-in kernels for common operations

Using RenderScript from Java Code

- Put data in Allocations
  - Use copy functions from `Allocation` class
- Set script globals
  - Using `set_globalname` from `ScriptC_filename` class
    - `set_elements(int)`
- Execute kernels
  - Using `forEach_kernelname()` from `ScriptC_filename` class
  - Takes one or two Allocations as arguments
  - Executed over the input entire Allocation by default
- Default root function
  - Invoked using `forEach_root`

Using RenderScript from Java Code

- Launch invoked functions
  - Using `invoke_functionname` from `ScriptC_filename` class
- Obtain data from `Allocations`
  - Copy data into Java buffers using `copy` methods from `Allocation` class
  - Synchronizes with asynchronous kernel
- Destroy RenderScript context
  - Using `destroy()` function
  - Or let it be garbage collected
  - Further use causes an exception

- Pre-defined scripts
- `ScriptIntrinsicBlend`
  - Kernels for blending two buffers
- `ScriptIntrinsicBlur`
  - Gaussian blur filter
  - Apply blur of a specified radius to the elements of an allocation
- `ScriptIntrinsicColorMatrix`
  - Apply color matrix to allocation
  - Hue filter
  - Each element is multiplied with a 4x4 color matrix
- `ScriptIntrinsicConvolve3x3`
  - Embossing filter
  - Apply 3x3 convolve to allocation

```
private void useBlurScript () {
  RenderScript mRS = RenderScript.create(this);
  Allocation input = Allocation.createFromBitmap(mRS,
                                    inputBitmap);
  Allocation output = Allocation.createFromBitmap(mRS,
                                    outputBitmap);
  ScriptIntrinsicBlur script = ScriptIntrinsicBlur.create(mRS,
                                    Element.U8_4(mRS));
  script.setRadius((float)24.5);
  script.setInput(input);
  script.forEach(output);
  output.copyTo(outputBitmap);
}
```

- The code is executed in a RenderScript runtime layer
- Runtime API - computation portable and scalable to the number of cores
- Code compiled into intermediate bytecode using LLVM compiler part of the Android build system
- Bytecode compiled just-in-time to machine code by another LLVM compiler on the device
- Machine code is optimized for that platform and cached

- Manage memory allocation requests
- Large number of math functions
  - Scalar and vector typed overloaded versions of common functions
  - Adding, multiplying, dot product, cross product
  - Atomic arithmetic and comparison functions
- Conversion functions for primitives, vectors, matrix, date and time
- Vector types for defining two-, three- and four-vectors
- Logging functions

- Set of classes generated by the Android build tools
- Allow access to RenderScript runtime from the Android framework
- Methods and constructors for allocating memory for the RenderScript code
- Reflected components:
  - Class `ScriptC_filename` for each script
    - Non-static functions
    - Non-static global variables
    - Get/set methods for each variable
    - For a const the set method is not generated
    - Global pointers
  - `Class ScriptField_structname` for each structure
    - An array of the struct
    - Allocate memory for one or more instances of the struct

- Functions are reflected into `ScriptC_filename` class
- Asynchronous -> cannot have return values
- When function is called, the call is queued and executed when possible
- To send a value back to Java use `rsSendToClient()`
- For function `void touch(float x, float y, float pressure, int id)` it generates code:

```
public void invoke_touch(float x, float y,
                         float pressure, int id) {
    FieldPacker touch_fp = new FieldPacker(16);
    touch_fp.addF32(x);
    touch_fp.addF32(y);
    touch_fp.addF32(pressure);
    touch_fp.addI32(id);
    invoke(mExportFuncIdx_touch, touch_fp);
}
```

- Variables are reflected into `ScriptC_filename` class
- Set/get methods are generated
- For `uint32_t index` are generated:

```
private long mExportVar_index;
public void set_index(long v){
    mExportVar_index = v;
    setVar(mExportVarIdx_index, v);
}

public long get_index(){
    return mExportVar_index;
}
```

- Structs are reflected into `ScriptField_structname` classes
- Class extends `android.renderscript.Script.FieldBase`
- Class includes:
  - A static nested class `Item` that includes the fields of struct
  - An Item array
  - Get/set methods for each field in the struct
    - `index` parameter to specify exact Item in the array
    - Setter has `copyNow` parameter - immediately sync memory to RenderScript runtime
  - Get/set methods for a certain Item in the array
  - Overloaded `ScriptField_structname(RenderScript rs, int count)`
    - `count` - number of elements in the array to allocate
  - `resize()` - expand allocated memory (array dimension)
  - `copyAll()` - sync memory to the RenderScript runtime

- Pointers reflected into `ScriptC_filename` class
- Pointer to struct or any RenderScript type
- Struct cannot contain pointers or nested arrays
- For `int32_t *index` is generated:

```
private Allocation mExportVar_index;
public void bind_index(Allocation v) {
    mExportVar_index = v;
    if (v == null) bindAllocation(null, mExportVarIdx_index);
    else bindAllocation(v, mExportVarIdx_index);
}
public Allocation get_index() {
    return mExportVar_index;
```

- Bind function - bind allocated memory in the VM to RenderScript runtime
- Cannot allocate memory in the script

- Apps run in the Android VM
- RenderScript code runs natively and needs to access the memory allocated in the VM
- Binding
    - Memory allocated in the VM is attached to the RenderScript runtime
    - Needed for dynamically allocated memory
    - Scripts cannot allocate memory explicitly
    - Statically allocated memory is created at compile time

Memory Allocation

- `Element` class
  - One cell of memory allocation
  - Basic element - any RenderScript data type (float, float4, etc)
  - Complex element - list of basic elements, created from structs
- `Type` class
  - Encapsulates the Element and a number of dimensions
  - Layout of memory - usually an array of Elements
- `Allocation` class
  - Performs actual allocation based on the `Type`
  - Sync is needed when memory is modified

- Non-static global variables
  - Allocated statically at compile time
  - No allocation in Java
  - Initialized by the RenderScript layer
  - Access them from Java using get/set methods in the reflected class
- Global pointers
  - Allocate memory dynamically in Java through the reflected class
  - Bind memory to the RenderScript runtime
  - Access memory from Java or from script

Allocate and Bind Dynamic Memory

- For pointers to structs call `ScriptField_structname` class constructor
- Call reflected `bind` method - bind memory to RenderScript runtime

```
ScriptField_Point points = new ScriptField_Point(mRS, 10);
mScript.bind_points(points);
```

- For primitive pointers - manually create `Allocation`

```
Allocation elements = Allocation.createSized(mRS,
                                   Element.I32(mRS), 10);
mScript.bind_elements(elements);
```

- Statically allocated memory
- Get/set methods in Java, access directly in script
- Changes in script are not propagated in Java
- Access in script:

```
typedef struct Point {
    int x;
    int y;
} Point_t;
Point_t point;
[..]
point.x = 1;
point.y = 1;
rsDebug("Point:", point.x, point.y);
```

Access global variables

- If you modify in Java, values are propagated to the RenderScript runtime
- Cannot get modifications from script
- Access in Java:

```
ScriptC_example mScript;
[..]
Item p = new ScriptField_Point.Item();
p.x = 1;
p.y = 1;
mScript.set_point(p);
Log.i("TAG", "Point: " + mScript.get_point().x
                 + " " + mScript.get_point().y);
```

- Dynamically allocated memory
- Allocate memory in Java and bind to the RenderScript runtime
- Use get/set methods to access from Java
- Access pointers directly from script
- Changes are automatically propagated to Java
- From script:

```
typedef struct Point {
    int x;
    int y;
} Point_t;
Point_t *point;
point[index].x = 1;
point[index].y = 1;
```

- From Java:

```
ScriptField_Point points = new ScriptField_Point(mRS, 10);
Item p = new ScriptField_Point.Item();
p.x = 25;
p.y = 70;
points.set(p, 0, true);
mScript.bind_point(points);
points.get_x(0);
points.get_y(0);
```

- Call bind just once
- No need to re-bind every time a change is made

- http://developer.android.com/guide/topics/renderscript/compute.html
- http://developer.android.com/guide/topics/renderscript/advanced.html
- Android Recipes A Problem-Solution Approach, Chapter 9

- RenderScript
- C99
- LLVM
- Allocation
- Compute kernels
- Invokable functions
- Reflected layer
- Pointers
- Structs
- Memory binding
- Element
- Type