Advanced RenderScript

In this document

Because applications that utilize RenderScript still run inside of the Android VM, you have access to all of the framework APIs that you are familiar with, but can utilize RenderScript when appropriate. To facilitate this interaction between the framework and the RenderScript runtime, an intermediate layer of code is also present to facilitate communication and memory management between the two levels of code. This document goes into more detail about these different layers of code as well as how memory is shared between the Android VM and RenderScript runtime.

RenderScript Runtime Layer

Your RenderScript code is compiled and executed in a compact and well-defined runtime layer. The RenderScript runtime APIs offer support for intensive computation that is portable and automatically scalable to the amount of cores available on a processor.

> **Note:** The standard C functions in the NDK must be guaranteed to run on a CPU, so RenderScript cannot access these libraries, because RenderScript is designed to run on different types of processors.

You define your RenderScript code in `.rs` and `.rsh` files in the `src/` directory of your Android project. The code is compiled to intermediate bytecode by the `llvm` compiler that runs as part of an Android build. When your application runs on a device, the bytecode is then compiled (just-in-time) to machine code by another `llvm` compiler that resides on the device. The machine code is optimized for the device and also cached, so subsequent uses of the RenderScript enabled application do not recompile the bytecode.

Some key features of the RenderScript runtime libraries include:

- Memory allocation request features
- A large collection of math functions with both scalar and vector typed overloaded versions of many common routines. Operations such as adding, multiplying, dot product, and cross product are available as well as atomic arithmetic and comparison functions.
- Conversion routines for primitive data types and vectors, matrix routines, and date and time routines
- Data types and structures to support the RenderScript system such as Vector types for defining two-, three-, or four-vectors.
- Logging functions

See the RenderScript runtime API reference for more information on the available functions.

Reflected Layer

The reflected layer is a set of classes that the Android build tools generate to allow access to the RenderScript runtime from the Android framework. This layer also provides methods and constructors that allow you to allocate and work with memory for pointers that are defined in your RenderScript code. The following list describes the major components that are reflected:

- Every `.rs` file that you create is generated into a class named `project_root/gen/package/name/ScriptC_renderscript_filename` of type `ScriptC`. This file is the `.java` version of your `.rs` file, which you can call from the Android framework. This class contains the following items reflected from the `.rs` file:
  - Non-static functions
  - Non-static, global RenderScript variables. Accessor methods are generated for each variable, so you can read and write the RenderScript variables from the Android framework. If a global variable is initialized at the RenderScript runtime layer, those values are used to initialize the corresponding values in the Android framework layer. If global variables are marked as `const`, then a `set` method is not generated. Look here for more details.
  - Global pointers
- A `struct` is reflected into its own class named `project_root/gen/package/name/ScriptField_struct_name`, which extends `Script.FieldBase`. This class represents an array of the `struct`, which allows you to allocate memory for one or more instances of this `struct`.

Functions

Functions are reflected into the script class itself, located in `project_root/gen/package/name/ScriptC_renderscript_filename`. For example, if you define the following function in your RenderScript code:

```
void touch(float x, float y, float pressure, int id) {
    if (id >= 10) {
        return;
    }

    touchPos[id].x = x;
    touchPos[id].y = y;
    touchPressure[id] = pressure;
}
```

then the following Java code is generated:

```
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);
}
```

Functions cannot have return values, because the RenderScript system is designed to be asynchronous. When your Android framework code calls into RenderScript, the call is queued and is executed when possible. This restriction allows the RenderScript system to function without constant interruption and increases efficiency. If functions were allowed to have return values, the call would block until the value was returned.

If you want the RenderScript code to send a value back to the Android framework, use the `rsSendToClient()` function.

Variables

Variables of supported types are reflected into the script class itself, located in `project_root/gen/package/name/ScriptC_renderscript_filename`. A set of accessor methods is generated for each variable. For example, if you define the following variable in your RenderScript code:

```
uint32_t unsignedInteger = 1;
```

then the following Java code is generated:

```
private long mExportVar_unsignedInteger;
public void set_unsignedInteger(long v){
    mExportVar_unsignedInteger = v;
    setVar(mExportVarIdx_unsignedInteger, v);
}

public long get_unsignedInteger(){
    return mExportVar_unsignedInteger;
}
```

Structs

Structs are reflected into their own classes, located in `<project_root>/gen/com/example/renderscript/ScriptField_struct_name`. This class represents an array of the `struct` and allows you to allocate memory for a specified number of `struct`s. For example, if you define the following struct:

```
typedef struct Point {
    float2 position;
    float size;
} Point_t;
```

then the following code is generated in `ScriptField_Point.java`:

```
package com.example.android.rs.hellocompute;

import android.renderscript.*;
import android.content.res.Resources;

  /**
  * @hide
  */
public class ScriptField_Point extends android.renderscript.Script.FieldBase {

    static public class Item {
        public static final int sizeof = 12;

        Float2 position;
        float size;

        Item() {
            position = new Float2();
        }
    }

    private Item mItemArray[];
    private FieldPacker mIOBuffer;
    public static Element createElement(RenderScript rs) {
        Element.Builder eb = new Element.Builder(rs);
        eb.add(Element.F32_2(rs), "position");
        eb.add(Element.F32(rs), "size");
        return eb.create();
    }

    public  ScriptField_Point(RenderScript rs, int count) {
        mItemArray = null;
        mIOBuffer = null;
        mElement = createElement(rs);
        init(rs, count);
    }

    public  ScriptField_Point(RenderScript rs, int count, int usages) {
        mItemArray = null;
        mIOBuffer = null;
        mElement = createElement(rs);
        init(rs, count, usages);
```

```
    }

    private void copyToArray(Item i, int index) {
        if (mIOBuffer == null) mIOBuffer = new FieldPacker(Item.sizeof * getType().getX()/* count
        */);
        mIOBuffer.reset(index * Item.sizeof);
        mIOBuffer.addF32(i.position);
        mIOBuffer.addF32(i.size);
    }

    public void set(Item i, int index, boolean copyNow) {
        if (mItemArray == null) mItemArray = new Item[getType().getX() /* count */];
        mItemArray[index] = i;
        if (copyNow)  {
            copyToArray(i, index);
            mAllocation.setFromFieldPacker(index, mIOBuffer);
        }
    }

    public Item get(int index) {
        if (mItemArray == null) return null;
        return mItemArray[index];
    }

    public void set_position(int index, Float2 v, boolean copyNow) {
        if (mIOBuffer == null) mIOBuffer = new FieldPacker(Item.sizeof * getType().getX()/* count */);
        if (mItemArray == null) mItemArray = new Item[getType().getX() /* count */];
        if (mItemArray[index] == null) mItemArray[index] = new Item();
        mItemArray[index].position = v;
        if (copyNow) {
            mIOBuffer.reset(index * Item.sizeof);
            mIOBuffer.addF32(v);
            FieldPacker fp = new FieldPacker(8);
            fp.addF32(v);
            mAllocation.setFromFieldPacker(index, 0, fp);
        }
    }

    public void set_size(int index, float v, boolean copyNow) {
        if (mIOBuffer == null) mIOBuffer = new FieldPacker(Item.sizeof * getType().getX()/* count */);
        if (mItemArray == null) mItemArray = new Item[getType().getX() /* count */];
        if (mItemArray[index] == null) mItemArray[index] = new Item();
        mItemArray[index].size = v;
        if (copyNow)  {
            mIOBuffer.reset(index * Item.sizeof + 8);
            mIOBuffer.addF32(v);
            FieldPacker fp = new FieldPacker(4);
            fp.addF32(v);
            mAllocation.setFromFieldPacker(index, 1, fp);
        }
    }

    public Float2 get_position(int index) {
        if (mItemArray == null) return null;
        return mItemArray[index].position;
    }

    public float get_size(int index) {
        if (mItemArray == null) return 0;
        return mItemArray[index].size;
    }

    public void copyAll() {
        for (int ct = 0; ct < mItemArray.length; ct++) copyToArray(mItemArray[ct], ct);
        mAllocation.setFromFieldPacker(0, mIOBuffer);
    }

    public void resize(int newSize) {
        if (mItemArray != null)  {
            int oldSize = mItemArray.length;
            int copySize = Math.min(oldSize, newSize);
            if (newSize == oldSize) return;
            Item ni[] = new Item[newSize];
```

```
            Item ni[] = new Item[newSize];
            System.arraycopy(mItemArray, 0, ni, 0, copySize);
            mItemArray = ni;
        }
        mAllocation.resize(newSize);
        if (mIOBuffer != null) mIOBuffer = new FieldPacker(Item.sizeof * getType().getX()/* count */);
    }
}
```

The generated code is provided to you as a convenience to allocate memory for structs requested by the RenderScript runtime and to interact with `structs` in memory. Each `struct`'s class defines the following methods and constructors:

- Overloaded constructors that allow you to allocate memory. The `ScriptField_struct_name(RenderScript rs, int count)` constructor allows you to define the number of structures that you want to allocate memory for with the `count` parameter. The `ScriptField_struct_name(RenderScript rs, int count, int usages)` constructor defines an extra parameter, `usages`, that lets you specify the memory space of this memory allocation. There are four memory space possibilities:
  - `USAGE_SCRIPT`: Allocates in the script memory space. This is the default memory space if you do not specify a memory space.
  - `USAGE_GRAPHICS_TEXTURE`: Allocates in the texture memory space of the GPU.
  - `USAGE_GRAPHICS_VERTEX`: Allocates in the vertex memory space of the GPU.
  - `USAGE_GRAPHICS_CONSTANTS`: Allocates in the constants memory space of the GPU that is used by the various program objects.

  You can specify multiple memory spaces by using the bitwise `OR` operator. Doing so notifies the RenderScript runtime that you intend on accessing the data in the specified memory spaces. The following example allocates memory for a custom data type in both the script and vertex memory spaces:

  ```
  ScriptField_Point touchPoints = new ScriptField_Point(myRenderScript, 2,
  Allocation.USAGE_SCRIPT | Allocation.USAGE_GRAPHICS_VERTEX);
  ```

- A static nested class, `Item`, allows you to create an instance of the `struct`, in the form of an object. This nested class is useful if it makes more sense to work with the `struct` in your Android code. When you are done manipulating the object, you can push the object to the allocated memory by calling `set(Item i, int index, boolean copyNow)` and setting the `Item` to the desired position in the array. The RenderScript runtime automatically has access to the newly written memory.
- Accessor methods to get and set the values of each field in a struct. Each of these accessor methods has an `index` parameter to specify the `struct` in the array that you want to read or write to. Each setter method also has a `copyNow` parameter that specifies whether or not to immediately sync this memory to the RenderScript runtime. To sync any memory that has not been synced, call `copyAll()`.
- The `createElement()` method creates a description of the struct in memory. This description is used to allocate memory consisting of one or many elements.
- `resize()` works much like a `realloc()` in C, allowing you to expand previously allocated memory, maintaining the current values that were previously created.
- `copyAll()` synchronizes memory that was set on the framework level to the RenderScript runtime. When you call a set accessor method on a member, there is an optional `copyNow` boolean parameter that you can specify. Specifying `true` synchronizes the memory when you call the method. If you specify false, you can call `copyAll()` once, and it synchronizes memory for all the properties that are not yet synchronized.

Pointers

Global pointers are reflected into the script class itself, located in `project_root/gen/package/name/ScriptC_renderscript_filename`. You can declare pointers to a `struct` or any of the supported RenderScript types, but a `struct` cannot contain pointers or nested arrays. For example, if you define the following pointers to a `struct` and `int32_t`

```
typedef struct Point {
    float2 position;
    float size;
} Point_t;

Point_t *touchPoints;
int32_t *intPointer;
```

then the following Java code is generated:

```
private ScriptField_Point mExportVar_touchPoints;
public void bind_touchPoints(ScriptField_Point v) {
    mExportVar_touchPoints = v;
    if (v == null) bindAllocation(null, mExportVarIdx_touchPoints);
    else bindAllocation(v.getAllocation(), mExportVarIdx_touchPoints);
}

public ScriptField_Point get_touchPoints() {
    return mExportVar_touchPoints;
}

private Allocation mExportVar_intPointer;
public void bind_intPointer(Allocation v) {
    mExportVar_intPointer = v;
    if (v == null) bindAllocation(null, mExportVarIdx_intPointer);
    else bindAllocation(v, mExportVarIdx_intPointer);
}

public Allocation get_intPointer() {
    return mExportVar_intPointer;
}
```

A `get` method and a special method named `bind_pointer_name` (instead of a `set()` method) are generated. The `bind_pointer_name` method allows you to bind the memory that is allocated in the Android VM to the RenderScript runtime (you cannot allocate memory in your `.rs` file). For more information, see Working with Allocated Memory.

Memory Allocation APIs

Applications that use RenderScript still run in the Android VM. The actual RenderScript code, however, runs natively and needs access to the memory allocated in the Android VM. To accomplish this, you must attach the memory that is allocated in the VM to the RenderScript runtime. This process, called binding, allows the RenderScript runtime to seamlessly work with memory that it requests but cannot explicitly allocate. The end result is essentially the same as if you had called `malloc` in C. The added benefit is that the Android VM can carry out garbage collection as well as share memory with the RenderScript runtime layer. Binding is only necessary for dynamically allocated memory. Statically allocated memory is automatically created for your RenderScript code at compile time. See Figure 1 for more information on how memory allocation occurs.

To support this memory allocation system, there are a set of APIs that allow the Android VM to allocate memory and offer similar functionality to a `malloc` call. These classes essentially describe how memory should be allocated and also carry out the allocation. To better understand how these classes work, it is useful to think of them in relation to a simple `malloc` call that can look like this:

```
array = (int *)malloc(sizeof(int)*10);
```

The `malloc` call can be broken up into two parts: the size of the memory being allocated (`sizeof(int)`), along with how many units of that memory should be allocated (10). The Android framework provides classes for these two parts as well as a class to represent `malloc` itself.

The `Element` class represents the (`sizeof(int)`) portion of the `malloc` call and encapsulates one cell of a memory allocation, such as a single float value or a struct. The `Type` class encapsulates the `Element` and the amount of elements to allocate (10 in our example). You can think of a `Type` as an array of `Element`s. The `Allocation` class does the actual memory allocation based on a given `Type` and represents the actual allocated memory.

In most situations, you do not need to call these memory allocation APIs directly. The reflected layer classes generate code to use these APIs

automatically and all you need to do to allocate memory is call a constructor that is declared in one of the reflected layer classes and then bind the resulting memory `Allocation` to the RenderScript. There are some situations where you would want to use these classes directly to allocate memory on your own, such as loading a bitmap from a resource or when you want to allocate memory for pointers to primitive types. You can see how to do this in the Allocating and binding memory to the RenderScript section. The following table describes the three memory management classes in more detail:

Android Object Type	Description
`Element`	An element describes one cell of a memory allocation and can have two forms: basic or complex. A basic element contains a single component of data of any valid RenderScript data type. Examples of basic element data types include a single `float` value, a `float4` vector, or a single RGB-565 color. Complex elements contain a list of basic elements and are created from `struct`s that you declare in your RenderScript code. For instance an allocation can contain multiple `struct`s arranged in order in memory. Each struct is considered as its own element, rather than each data type within that struct.
`Type`	A type is a memory allocation template and consists of an element and one or more dimensions. It describes the layout of the memory (basically an array of `Element`s) but does not allocate the memory for the data that it describes. A type consists of five dimensions: X, Y, Z, LOD (level of detail), and Faces (of a cube map). You can set the X,Y,Z dimensions to any positive integer value within the constraints of available memory. A single dimension allocation has an X dimension of greater than zero while the Y and Z dimensions are zero to indicate not present. For example, an allocation of x=10, y=1 is considered two dimensional and x=10, y=0 is considered one dimensional. The LOD and Faces dimensions are booleans to indicate present or not present.
`Allocation`	An allocation provides the memory for applications based on a description of the memory that is represented by a `Type`. Allocated memory can exist in many memory spaces concurrently. If memory is modified in one space, you must explicitly synchronize the memory, so that it is updated in all the other spaces in which it exists. Allocation data is uploaded in one of two primary ways: type checked and type unchecked. For simple arrays there are `copyFrom()` functions that take an array from the Android system and copy it to the native layer memory store. The unchecked variants allow the Android system to copy over arrays of structures because it does not support structures. For example, if there is an allocation that is an array of n floats, the data contained in a float[n] array or a `byte[n*4]` array can be copied.

Working with Memory

Non-static, global variables that you declare in your RenderScript are allocated memory at compile time. You can work with these variables directly in your RenderScript code without having to allocate memory for them at the Android framework level. The Android framework layer also has access to these variables with the provided accessor methods that are generated in the reflected layer classes. If these variables are initialized at the RenderScript runtime layer, those values are used to initialize the corresponding values in the Android framework layer. If global variables are marked as const, then a `set` method is not generated. Look here for more details.

> **Note:** If you are using certain RenderScript structures that contain pointers, such as `rs_program_fragment` and `rs_allocation`, you have to obtain an object of the corresponding Android framework class first and then call the `set` method for that structure to bind the memory to the RenderScript runtime. You cannot directly manipulate these structures at the RenderScript runtime layer. This restriction is not applicable to user-defined structures that contain pointers, because they cannot be exported to a reflected layer class in the first place. A compiler error is generated if you try to declare a non-static, global struct that contains a pointer.

RenderScript also has support for pointers, but you must explicitly allocate the memory in your Android framework code. When you declare a global pointer in your `.rs` file, you allocate memory through the appropriate reflected layer class and bind that memory to the native RenderScript layer. You can interact with this memory from the Android framework layer as well as the RenderScript layer, which offers you the flexibility to modify variables in the most appropriate layer.

Allocating and binding dynamic memory to the RenderScript

To allocate dynamic memory, you need to call the constructor of a `Script.FieldBase` class, which is the most common way. An alternative is to create an `Allocation` manually, which is required for things such as primitive type pointers. You should use a `Script.FieldBase` class constructor whenever available for simplicity. After obtaining a memory allocation, call the reflected `bind` method of the pointer to bind the allocated memory to the RenderScript runtime.

The example below allocates memory for both a primitive type pointer, `intPointer`, and a pointer to a struct, `touchPoints`. It also binds the memory to the RenderScript:

```
private RenderScript myRenderScript;
private ScriptC_example script;
private Resources resources;

public void init(RenderScript rs, Resources res) {
    myRenderScript = rs;
    resources = res;

    //allocate memory for the struct pointer, calling the constructor
    ScriptField_Point touchPoints = new ScriptField_Point(myRenderScript, 2);

    //Create an element manually and allocate memory for the int pointer
    intPointer = Allocation.createSized(myRenderScript, Element.I32(myRenderScript), 2);

    //create an instance of the RenderScript, pointing it to the bytecode resource
    mScript = new ScriptC_example(myRenderScript, resources, R.raw.example);

    //bind the struct and int pointers to the RenderScript
    mScript.bind_touchPoints(touchPoints);
    script.bind_intPointer(intPointer);

   ...
}
```

Reading and writing to memory

You can read and write to statically and dynamically allocated memory both at the RenderScript runtime and Android framework layer.

Statically allocated memory comes with a one-way communication restriction at the RenderScript runtime level. When RenderScript code changes the value of a variable, it is not communicated back to the Android framework layer for efficiency purposes. The last value that is set from the Android framework is always returned during a call to a `get` method. However, when Android framework code modifies a variable, that change can be communicated to the RenderScript runtime automatically or synchronized at a later time. If you need to send data from the RenderScript runtime to the Android framework layer, you can use the `rsSendToClient()` function to overcome this limitation.

When working with dynamically allocated memory, any changes at the RenderScript runtime layer are propagated back to the Android framework layer if you modified the memory allocation using its associated pointer. Modifying an object at the Android framework layer immediately propagates that change back to the RenderScript runtime layer.

Reading and writing to global variables

Reading and writing to global variables is a straightforward process. You can use the accessor methods at the Android framework level or set them directly in the RenderScript code. Keep in mind that any changes that you make in your RenderScript code are not propagated back to the Android framework layer (look here for more details).

For example, given the following struct declared in a file named `rsfile.rs`:

```
typedef struct Point {
    int x;
    int y;
} Point_t;

Point_t point;
```

You can assign values to the struct like this directly in `rsfile.rs`. These values are not propagated back to the Android framework level:

```
point.x = 1;
point.y = 1;
```

You can assign values to the struct at the Android framework layer like this. These values are propagated back to the RenderScript runtime level asynchronously:

```
ScriptC_rsfile mScript;

...

Item i = new ScriptField_Point.Item();
i.x = 1;
i.y = 1;
mScript.set_point(i);
```

You can read the values in your RenderScript code like this:

```
rsDebug("Printing out a Point", point.x, point.y);
```

You can read the values in the Android framework layer with the following code. Keep in mind that this code only returns a value if one was set at the Android framework level. You will get a null pointer exception if you only set the value at the RenderScript runtime level:

```
Log.i("TAGNAME", "Printing out a Point: " + mScript.get_point().x + " " + mScript.get_point().y);
System.out.println(point.get_x() + " " + point.get_y());
```

Reading and writing global pointers

Assuming that memory has been allocated in the Android framework level and bound to the RenderScript runtime, you can read and write memory from the Android framework level by using the `get` and `set` methods for that pointer. In the RenderScript runtime layer, you can read and write to memory with pointers as normal and the changes are propagated back to the Android framework layer, unlike with statically allocated memory.

For example, given the following pointer to a `struct` in a file named `rsfile.rs`:

```
typedef struct Point {
    int x;
    int y;
} Point_t;

Point_t *point;
```

Assuming you already allocated memory at the Android framework layer, you can access values in the `struct` as normal. Any changes you make to the struct via its pointer variable are automatically available to the Android framework layer:

```
point[index].x = 1;
point[index].y = 1;
```

You can read and write values to the pointer at the Android framework layer as well:

```
ScriptField_Point p = new ScriptField_Point(mRS, 1);
    Item i = new ScriptField_Point.Item();
    i.x=100;
    i.y = 100;
    p.set(i, 0, true);
    mScript.bind_point(p);

    points.get_x(0);            //read x and y from index 0
    points.get_x(0);
```

Once memory is already bound, you do not have to rebind the memory to the RenderScript runtime every time you make a change to a value.