RenderScript Runtime API Reference

Overview

RenderScript is a high-performance runtime that provides compute operations at the native level. RenderScript code is compiled on devices at runtime to allow platform-independence as well.

This reference documentation describes the RenderScript runtime APIs, which you can utilize to write RenderScript code in C99. The RenderScript compute header files are automatically included for you.

To use RenderScript, you need to utilize the RenderScript runtime APIs documented here as well as the Android framework APIs for RenderScript. For documentation on the Android framework APIs, see the android.renderscript package reference.

For more information on how to develop with RenderScript and how the runtime and Android framework APIs interact, see the RenderScript developer guide and the RenderScript samples.

Numerical Types

Scalars:

RenderScript supports the following scalar numerical types:

	8 bits	16 bits	32 bits	64 bits
Integer:	char, int8_t	short, int16_t	int32_t	long, long long, int64_t
Unsigned integer:	uchar, uint8_t	ushort, uint16_t	uint, uint32_t	ulong, uint64_t
Floating point:		half	float	double

Vectors:

RenderScript supports fixed size vectors of length 2, 3, and 4. Vectors are declared using the common type name followed by a 2, 3, or 4. E.g. float4, int3, double2, ulong4.

To create vector literals, use the vector type followed by the values enclosed between curly braces, e.g. `(float3){1.0f, 2.0f, 3.0f}`.

Entries of a vector can be accessed using different naming styles.

Single entries can be accessed by following the variable name with a dot and:

- The letters x, y, z, and w,
- The letters r, g, b, and a,
- The letter s or S, followed by a zero based index.

For example, with `int4 myVar;` the following are equivalent:

```
myVar.x == myVar.r == myVar.s0 == myVar.S0
myVar.y == myVar.g == myVar.s1 == myVar.S1
myVar.z == myVar.b == myVar.s2 == myVar.S2
myVar.w == myVar.a == myVar.s3 == myVar.S3
```

Multiple entries of a vector can be accessed at once by using an identifier that is the concatenation of multiple letters or indices. The resulting vector has a size equal to the number of entries named.

With the example above, the middle two entries can be accessed using `myVar.yz`, `myVar.gb`, `myVar.s12`, and `myVar.S12`.

The entries don't have to be contiguous or in increasing order. Entries can even be repeated, as long as we're not trying to assign to it. You also can't mix the naming styles.

Here are examples of what can or can't be done:

```
float4 v4;
float3 v3;
float2 v2;
v2 = v4.xx; // Valid
v3 = v4.zxw; // Valid
v3 = v4.bba; // Valid
v3 = v4.s032; // Valid
v3.s120 = v4.S233; // Valid
v4.yz = v3.rg; // Valid
v4.yzx = v3.rg; // Invalid: mismatched sizes
v4.yzz = v3; // Invalid: z appears twice in an assignment
v3 = v3.xas0; // Invalid: can't mix xyzw with rgba nor s0...
v3 = v4.s034; // Invalid: the digit can only be 0, 1, 2, or 3
```

Matrices and Quaternions:

RenderScript supports fixed size square matrices of floats of size 2x2, 3x3, and 4x4. The types are named rs_matrix2x2, rs_matrix3x3, and rs_matrix4x4. See Matrix Functions for the list of operations.

Quaternions are also supported via rs_quaternion. See Quaterion Functions for the list of operations.

Types	
char2	Two 8 bit signed integers
char3	Three 8 bit signed integers
char4	Four 8 bit signed integers
double2	Two 64 bit floats
double3	Three 64 bit floats
double4	Four 64 bit floats
float2	Two 32 bit floats
float3	Three 32 bit floats
float4	Four 32 bit floats
half	16 bit floating point value
half2	Two 16 bit floats
half3	Three 16 bit floats
half4	Four 16 bit floats
int16_t	16 bit signed integer
int2	Two 32 bit signed integers
int3	Three 32 bit signed integers
int32_t	32 bit signed integer
int4	Four 32 bit signed integers
int64_t	64 bit signed integer
int8_t	8 bit signed integer
long2	Two 64 bit signed integers

long3	Three 64 bit signed integers
long4	Four 64 bit signed integers
rs_matrix2x2	2x2 matrix of 32 bit floats
rs_matrix3x3	3x3 matrix of 32 bit floats
rs_matrix4x4	4x4 matrix of 32 bit floats
rs_quaternion	Quaternion
short2	Two 16 bit signed integers
short3	Three 16 bit signed integers
short4	Four 16 bit signed integers
size_t	Unsigned size type
ssize_t	Signed size type
uchar	8 bit unsigned integer
uchar2	Two 8 bit unsigned integers
uchar3	Three 8 bit unsigned integers
uchar4	Four 8 bit unsigned integers
uint	32 bit unsigned integer
uint16_t	16 bit unsigned integer
uint2	Two 32 bit unsigned integers
uint3	Three 32 bit unsigned integers
uint32_t	32 bit unsigned integer
uint4	Four 32 bit unsigned integers
uint64_t	64 bit unsigned integer
uint8_t	8 bit unsigned integer
ulong	64 bit unsigned integer
ulong2	Two 64 bit unsigned integers
ulong3	Three 64 bit unsigned integers
ulong4	Four 64 bit unsigned integers
ushort	16 bit unsigned integer
ushort2	Two 16 bit unsigned integers
ushort3	Three 16 bit unsigned integers
ushort4	Four 16 bit unsigned integers

Object Types

The types below are used to manipulate RenderScript objects like allocations, samplers, elements, and scripts. Most of these object are created using the Java RenderScript APIs.

Types	
rs_allocation	Handle to an allocation
rs_allocation_cubemap_face	Enum for selecting cube map faces
rs_allocation_usage_type	Bitfield to specify how an allocation is used

rs_data_kind	Element data kind
rs_data_type	Element basic data type
rs_element	Handle to an element
rs_sampler	Handle to a Sampler
rs_sampler_value	Sampler wrap T value
rs_script	Handle to a Script
rs_type	Handle to a Type
rs_yuv_format	YUV format

Conversion Functions

The functions below convert from a numerical vector type to another, or from one color representation to another.

Functions	
convert	Convert numerical vectors
rsPackColorTo8888	Create a uchar4 RGBA from floats
rsUnpackColor8888	Create a float4 RGBA from uchar4
rsYuvToRGBA	Convert a YUV value to RGBA

Mathematical Constants and Functions

The mathematical functions below can be applied to scalars and vectors. When applied to vectors, the returned value is a vector of the function applied to each entry of the input.

For example:

```
float3 a, b;
// The following call sets
// a.x to sin(b.x),
// a.y to sin(b.y), and
// a.z to sin(b.z).
a = sin(b);
```

See Vector Math Functions for functions like distance() and length() that interpret instead the input as a single vector in n-dimensional space.

The precision of the mathematical operations on 32 bit floats is affected by the pragmas rs_fp_relaxed and rs_fp_full. Under rs_fp_relaxed, subnormal values may be flushed to zero and rounding may be done towards zero. In comparison, rs_fp_full requires correct handling of subnormal values, i.e. smaller than 1.17549435e-38f. rs_fp_rull also requires round to nearest with ties to even.

Different precision/speed tradeoffs can be achieved by using variants of the common math functions. Functions with a name starting with

- native_: May have custom hardware implementations with weaker precision. Additionally, subnormal values may be flushed to zero, rounding towards zero may be used, and NaN and infinity input may not be handled correctly.
- half_: May perform internal computations using 16 bit floats. Additionally, subnormal values may be flushed to zero, and rounding towards zero may be used.

Constants	
M_1_PI	1 / pi, as a 32 bit float
M_2_PI	2 / pi, as a 32 bit float
M_2_SQRTPI	2 / sqrt(pi), as a 32 bit float
M_E	e, as a 32 bit float

M_LN10	log_e(10), as a 32 bit float
M_LN2	log_e(2), as a 32 bit float
M_LOG10E	log_10(e), as a 32 bit float
M_LOG2E	log_2(e), as a 32 bit float
M_PI	pi, as a 32 bit float
M_PI_2	pi / 2, as a 32 bit float
M_PI_4	pi / 4, as a 32 bit float
M_SQRT1_2	1 / sqrt(2), as a 32 bit float
M_SQRT2	sqrt(2), as a 32 bit float

Functions	
abs	Absolute value of an integer
acos	Inverse cosine
acosh	Inverse hyperbolic cosine
acospi	Inverse cosine divided by pi
asin	Inverse sine
asinh	Inverse hyperbolic sine
asinpi	Inverse sine divided by pi
atan	Inverse tangent
atan2	Inverse tangent of a ratio
atan2pi	Inverse tangent of a ratio, divided by pi
atanh	Inverse hyperbolic tangent
atanpi	Inverse tangent divided by pi
cbrt	Cube root
ceil	Smallest integer not less than a value
clamp	Restrain a value to a range
clz	Number of leading 0 bits
copysign	Copies the sign of a number to another
cos	Cosine
cosh	Hypebolic cosine
cospi	Cosine of a number multiplied by pi
degrees	Converts radians into degrees
erf	Mathematical error function
erfc	Mathematical complementary error function
exp	e raised to a number
exp10	10 raised to a number
exp2	2 raised to a number
expm1	e raised to a number minus one
fabs	Absolute value of a float
fdim	Positive difference between two values

floor	Smallest integer not greater than a value
fma	Multiply and add
fmax	Maximum of two floats
fmin	Minimum of two floats
fmod	Modulo
fract	Positive fractional part
frexp	Binary mantissa and exponent
half_recip	Reciprocal computed to 16 bit precision
half_rsqrt	Reciprocal of a square root computed to 16 bit precision
half_sqrt	Square root computed to 16 bit precision
hypot	Hypotenuse
ilogb	Base two exponent
ldexp	Creates a floating point from mantissa and exponent
lgamma	Natural logarithm of the gamma function
log	Natural logarithm
log10	Base 10 logarithm
log1p	Natural logarithm of a value plus 1
log2	Base 2 logarithm
logb	Base two exponent
mad	Multiply and add
max	Maximum
min	Minimum
mix	Mixes two values
modf	Integral and fractional components
nan	Not a Number
nan_half	Not a Number
native_acos	Approximate inverse cosine
native_acosh	Approximate inverse hyperbolic cosine
native_acospi	Approximate inverse cosine divided by pi
native_asin	Approximate inverse sine
native_asinh	Approximate inverse hyperbolic sine
native_asinpi	Approximate inverse sine divided by pi
native_atan	Approximate inverse tangent
native_atan2	Approximate inverse tangent of a ratio
native_atan2pi	Approximate inverse tangent of a ratio, divided by pi
native_atanh	Approximate inverse hyperbolic tangent
native_atanpi	Approximate inverse tangent divided by pi
native_cbrt	Approximate cube root
native_cos	Approximate cosine
native_cosh	Approximate hypebolic cosine

native_cospi	Approximate cosine of a number multiplied by pi
native_divide	Approximate division
native_exp	Approximate e raised to a number
native_exp10	Approximate 10 raised to a number
native_exp2	Approximate 2 raised to a number
native_expm1	Approximate e raised to a number minus one
native_hypot	Approximate hypotenuse
native_log	Approximate natural logarithm
native_log10	Approximate base 10 logarithm
native_log1p	Approximate natural logarithm of a value plus 1
native_log2	Approximate base 2 logarithm
native_powr	Approximate positive base raised to an exponent
native_recip	Approximate reciprocal
native_rootn	Approximate nth root
native_rsqrt	Approximate reciprocal of a square root
native_sin	Approximate sine
native_sincos	Approximate sine and cosine
native_sinh	Approximate hyperbolic sine
native_sinpi	Approximate sine of a number multiplied by pi
native_sqrt	Approximate square root
native_tan	Approximate tangent
native_tanh	Approximate hyperbolic tangent
native_tanpi	Approximate tangent of a number multiplied by pi
nextafter	Next floating point number
pow	Base raised to an exponent
pown	Base raised to an integer exponent
powr	Positive base raised to an exponent
radians	Converts degrees into radians
remainder	Remainder of a division
remquo	Remainder and quotient of a division
rint	Round to even
rootn	Nth root
round	Round away from zero
rsRand	Pseudo-random number
rsqrt	Reciprocal of a square root
sign	Sign of a value
sin	Sine
sincos	Sine and cosine
sinh	Hyperbolic sine

sinpi	Sine of a number multiplied by pi
sqrt	Square root
step	0 if less than a value, 0 otherwise
tan	Tangent
tanh	Hyperbolic tangent
tanpi	Tangent of a number multiplied by pi
tgamma	Gamma function
trunc	Truncates a floating point

Vector Math Functions

These functions interpret the input arguments as representation of vectors in n-dimensional space.

The precision of the mathematical operations on 32 bit floats is affected by the pragmas rs_fp_relaxed and rs_fp_full. See Mathematical Constants and Functions for details.

Different precision/speed tradeoffs can be achieved by using variants of the common math functions. Functions with a name starting with

- native_: May have custom hardware implementations with weaker precision. Additionally, subnormal values may be flushed to zero, rounding towards zero may be used, and NaN and infinity input may not be handled correctly.
- fast_: May perform internal computations using 16 bit floats. Additionally, subnormal values may be flushed to zero, and rounding towards zero may be used.

Functions	
cross	Cross product of two vectors
distance	Distance between two points
dot	Dot product of two vectors
fast_distance	Approximate distance between two points
fast_length	Approximate length of a vector
fast_normalize	Approximate normalized vector
length	Length of a vector
native_distance	Approximate distance between two points
native_length	Approximate length of a vector
native_normalize	Approximately normalize a vector
normalize	Normalize a vector

Matrix Functions

These functions let you manipulate square matrices of rank 2x2, 3x3, and 4x4. They are particularly useful for graphical transformations and are compatible with OpenGL.

We use a zero-based index for rows and columns. E.g. the last element of a rs_matrix4x4 is found at (3, 3).

RenderScript uses column-major matrices and column-based vectors. Transforming a vector is done by postmultiplying the vector, e.g. `(matrix * vector)`, as provided by rsMatrixMultiply().

To create a transformation matrix that performs two transformations at once, multiply the two source matrices, with the first transformation as the right argument. E.g. to create a transformation matrix that applies the transformation s1 followed by s2, call `rsMatrixLoadMultiply(&combined, &s2, &s1)`. This derives from `s2 * (s1 * v)`, which is `(s2 * s1) * v`.

We have two style of functions to create transformation matrices: rsMatrixLoad*Transformation* and rsMatrix*Transformation*. The former style simply stores the transformation matrix in the first argument. The latter modifies a pre-existing transformation matrix so that the new transformation happens first. E.g. if you call rsMatrixTranslate() on a matrix that already does a scaling, the resulting matrix when applied to a vector will first do the translation then the scaling.

Functions	
rsExtractFrustumPlanes	Compute frustum planes
rsIsSphereInFrustum	Checks if a sphere is within the frustum planes
rsMatrixGet	Get one element
rsMatrixInverse	Inverts a matrix in place
rsMatrixInverseTranspose	Inverts and transpose a matrix in place
rsMatrixLoad	Load or copy a matrix
rsMatrixLoadFrustum	Load a frustum projection matrix
rsMatrixLoadIdentity	Load identity matrix
rsMatrixLoadMultiply	Multiply two matrices
rsMatrixLoadOrtho	Load an orthographic projection matrix
rsMatrixLoadPerspective	Load a perspective projection matrix
rsMatrixLoadRotate	Load a rotation matrix
rsMatrixLoadScale	Load a scaling matrix
rsMatrixLoadTranslate	Load a translation matrix
rsMatrixMultiply	Multiply a matrix by a vector or another matrix
rsMatrixRotate	Apply a rotation to a transformation matrix
rsMatrixScale	Apply a scaling to a transformation matrix
rsMatrixSet	Set one element
rsMatrixTranslate	Apply a translation to a transformation matrix
rsMatrixTranspose	Transpose a matrix place

Quaternion Functions

The following functions manipulate quaternions.

Functions	
rsQuaternionAdd	Add two quaternions
rsQuaternionConjugate	Conjugate a quaternion
rsQuaternionDot	Dot product of two quaternions
rsQuaternionGetMatrixUnit	Get a rotation matrix from a quaternion
rsQuaternionLoadRotate	Create a rotation quaternion
rsQuaternionLoadRotateUnit	Quaternion that represents a rotation about an arbitrary unit vector
rsQuaternionMultiply	Multiply a quaternion by a scalar or another quaternion
rsQuaternionNormalize	Normalize a quaternion
rsQuaternionSet	Create a quaternion
rsQuaternionSlerp	Spherical linear interpolation between two quaternions

Atomic Update Functions

To update values shared between multiple threads, use the functions below. They ensure that the values are atomically updated, i.e. that the memory reads, the updates, and the memory writes are done in the right order.

These functions are slower than their non-atomic equivalents, so use them only when synchronization is needed.

Note that in RenderScript, your code is likely to be running in separate threads even though you did not explicitely create them. The RenderScript runtime will very often split the execution of one kernel across multiple threads. Updating globals should be done with atomic functions. If possible, modify your algorithm to avoid them altogether.

Functions	
rsAtomicAdd	Thread-safe addition
rsAtomicAnd	Thread-safe bitwise and
rsAtomicCas	Thread-safe compare and set
rsAtomicDec	Thread-safe decrement
rsAtomicInc	Thread-safe increment
rsAtomicMax	Thread-safe maximum
rsAtomicMin	Thread-safe minimum
rsAtomicOr	Thread-safe bitwise or
rsAtomicSub	Thread-safe subtraction
rsAtomicXor	Thread-safe bitwise exclusive or

Time Functions and Types

The functions below can be used to tell the current clock time and the current system up time. It is not recommended to call these functions inside of a kernel.

Types	
rs_time_t	Seconds since January 1, 1970
rs_tm	Date and time structure

Functions	
rsGetDt	Elapsed time since last call
rsLocaltime	Convert to local time
rsTime	Seconds since January 1, 1970
rsUptimeMillis	System uptime in milliseconds
rsUptimeNanos	System uptime in nanoseconds

Allocation Creation Functions

The functions below can be used to create Allocations from a Script.

These functions can be called directly or indirectly from an invokable function. If some control-flow path can result in a call to these functions from a RenderScript kernel function, a compiler error will be generated.

Functions	
rsCreateAllocation	Create an rs_allocation object of given Type.

rsCreateElement	Creates an rs_element object of the specified data type
rsCreatePixelElement	Creates an rs_element object of the specified data type and data kind
rsCreateType	Creates an rs_type object with the specified Element and shape attributes
rsCreateVectorElement	Creates an rs_element object of the specified data type and vector width

Allocation Data Access Functions

The functions below can be used to get and set the cells that comprise an allocation.

- Individual cells are accessed using the rsGetElementAt* and rsSetElementAt functions.
- Multiple cells can be copied using the rsAllocationCopy* and rsAllocationV* functions.
- For getting values through a sampler, use rsSample.

The rsGetElementAt and rsSetElement* functions are somewhat misnamed. They don't get or set elements, which are akin to data types; they get or set cells. Think of them as rsGetCellAt and and rsSetCellAt.

Functions	
rsAllocationCopy1DRange	Copy consecutive cells between allocations
rsAllocationCopy2DRange	Copy a rectangular region of cells between allocations
rsAllocationVLoadX	Get a vector from an allocation of scalars
rsAllocationVStoreX	Store a vector into an allocation of scalars
rsGetElementAt	Return a cell from an allocation
rsGetElementAtYuv_uchar_U	Get the U component of an allocation of YUVs
rsGetElementAtYuv_uchar_V	Get the V component of an allocation of YUVs
rsGetElementAtYuv_uchar_Y	Get the Y component of an allocation of YUVs
rsSample	Sample a value from a texture allocation
rsSetElementAt	Set a cell of an allocation

Object Characteristics Functions

The functions below can be used to query the characteristics of an Allocation, Element, or Sampler object. These objects are created from Java. You can't create them from a script.

Allocations:

Allocations are the primary method used to pass data to and from RenderScript kernels.

They are a structured collection of cells that can be used to store bitmaps, textures, arbitrary data points, etc.

This collection of cells may have many dimensions (X, Y, Z, Array0, Array1, Array2, Array3), faces (for cubemaps), and level of details (for mipmapping).

See the android.renderscript.Allocation for details on to create Allocations.

Elements:

The term "element" is used a bit ambiguously in RenderScript, as both type information for the cells of an Allocation and the instantiation of that type. For example:

- rs_element is a handle to a type specification, and
- In functions like rsGetElementAt(), "element" means the instantiation of the type, i.e. a cell of an Allocation.

The functions below let you query the characteristics of the type specificiation.

An Element can specify a simple data types as found in C, e.g. an integer, float, or boolean. It can also specify a handle to a RenderScript object. See rs_data_type for a list of basic types.

Elements can specify fixed size vector (of size 2, 3, or 4) versions of the basic types. Elements can be grouped together into complex Elements, creating the equivalent of C structure definitions.

Elements can also have a kind, which is semantic information used to interpret pixel data. See rs_data_kind.

When creating Allocations of common elements, you can simply use one of the many predefined Elements like F32_2.

To create complex Elements, use the Element.Builder Java class.

Samplers:

Samplers objects define how Allocations can be read as structure within a kernel. See android.renderscript.S.

Functions	
rsAllocationGetDimFaces	Presence of more than one face
rsAllocationGetDimLOD	Presence of levels of detail
rsAllocationGetDimX	Size of the X dimension
rsAllocationGetDimY	Size of the Y dimension
rsAllocationGetDimZ	Size of the Z dimension
rsAllocationGetElement	Get the object that describes the cell of an Allocation
rsClearObject	Release an object
rsElementGetBytesSize	Size of an Element
rsElementGetDataKind	Kind of an Element
rsElementGetDataType	Data type of an Element
rsElementGetSubElement	Sub-element of a complex Element
rsElementGetSubElementArraySize	Array size of a sub-element of a complex Element
rsElementGetSubElementCount	Number of sub-elements
rsElementGetSubElementName	Name of a sub-element
rsElementGetSubElementNameLength	Length of the name of a sub-element
rsElementGetSubElementOffsetBytes	Offset of the instantiated sub-element
rsElementGetVectorSize	Vector size of the Element
rsIsObject	Check for an empty handle
rsSamplerGetAnisotropy	Anisotropy of the Sampler
rsSamplerGetMagnification	Sampler magnification value
rsSamplerGetMinification	Sampler minification value
rsSamplerGetWrapS	Sampler wrap S value
rsSamplerGetWrapT	Sampler wrap T value

Kernel Invocation Functions and Types

The rsForEach() function can be used to invoke the root kernel of a script.

The other functions are used to get the characteristics of the invocation of an executing kernel, like dimensions and current indices. These functions take a rs_kernel_context as argument.

Types	
rs_for_each_strategy_t	Suggested cell processing order
rs_kernel	Handle to a kernel function
rs_kernel_context	Handle to a kernel invocation context
rs_script_call_t	Cell iteration information

Functions	
rsForEach	Launches a kernel
rsForEachInternal	(Internal API) Launch a kernel in the current Script (with the slot number)
rsForEachWithOptions	Launches a kernel with options
rsGetArray0	Index in the Array0 dimension for the specified kernel context
rsGetArray1	Index in the Array1 dimension for the specified kernel context
rsGetArray2	Index in the Array2 dimension for the specified kernel context
rsGetArray3	Index in the Array3 dimension for the specified kernel context
rsGetDimArray0	Size of the Array0 dimension for the specified kernel context
rsGetDimArray1	Size of the Array1 dimension for the specified kernel context
rsGetDimArray2	Size of the Array2 dimension for the specified kernel context
rsGetDimArray3	Size of the Array3 dimension for the specified kernel context
rsGetDimHasFaces	Presence of more than one face for the specified kernel context
rsGetDimLod	Number of levels of detail for the specified kernel context
rsGetDimX	Size of the X dimension for the specified kernel context
rsGetDimY	Size of the Y dimension for the specified kernel context
rsGetDimZ	Size of the Z dimension for the specified kernel context
rsGetFace	Coordinate of the Face for the specified kernel context
rsGetLod	Index in the Levels of Detail dimension for the specified kernel context

Input/Output Functions

These functions are used to:

- Send information to the Java client, and
- Send the processed allocation or receive the next allocation to process.

Functions	
rsAllocationIoReceive	Receive new content from the queue
rsAllocationIoSend	Send new content to the queue
rsSendToClient	Send a message to the client, non-blocking
rsSendToClientBlocking	Send a message to the client, blocking

Debugging Functions

The functions below are intended to be used during application developement. They should not be used in shipping applications.

Functions

rsDebug	Log a message and values

Graphics Functions and Types

The graphics subsystem of RenderScript was removed at API level 23.