Vendor: NVIDIA

Exam Code: NCP-OUSD

Exam Name: OpenUSD Development

Version: DEMO

QUESTION 1
If you have a Usd.Prim object named my_prim and you want to specifically retrieve an attribute named "size", which method would you typically use?

A. my_prim.GetAttribute("size")
B. my_prim.GetRelationship("size")
C. my_prim.GetProperty("size")
D. my_prim.GetPrimvar("size")

Answer: A
Explanation:
The correct method is my_prim.GetAttribute("size") because the question asks for a specific attribute, not a relationship or generic property. NVIDIA's Learn OpenUSD material defines properties as the data-bearing namespace objects on prims and distinguishes two property categories: attributes and relationships. Attributes hold typed values, while relationships point to other objects in the scene description. NVIDIA's Omniverse developer reference also states that Usd.Prim.GetAttribute() returns a Usd.Attribute, after which Usd.Attribute.Get() is used to resolve the actual authored or composed value. Option A is therefore the precise API call.

QUESTION 2
When USD computes the position of a PointInstancer instance, what additional position-related information is used beyond the position information for a given instance?

A. USD also uses the transform of the PointInstancer prim, if present, but ignores any transforms set on the prototype root.
B. USD also uses the transform of the root of the prototype and the transform of the PointInstancer prim itself, if present.
C. USD uses position information for the instance, and ignores any position information in the prototype root, or the PointInstancer prim itself.

Answer: B
Explanation:
A PointInstancer instance position is not computed from the per-instance positions array alone. NVIDIA's Learn OpenUSD point-instancing lesson explains that authored positions are in the local coordinate space of the PointInstancer. For the final world-space transform, USD combines multiple transforms: first the prototype root transform, then the instance-specific transform components such as scales[i], orientations[i], and positions[i], and finally the transform authored on the PointInstancer prim itself.

Option B is correct because it includes both additional transform sources that affect the resulting placement: the prototype root's transform and the PointInstancer prim's transform. This matters in production asset aggregation because prototypes may carry their own local placement or orientation, while the instancer prim may be moved, rotated, or scaled as a whole.

QUESTION 3
What is the correct prim type in UsdShade for sharing reusable portions of shading networks, allowing for parameterization?

A. SubNetwork
B. Custom Schema
C. NodeGraph

Answer: C
Explanation:
The correct UsdShade prim type is NodeGraph. NVIDIA's Learn OpenUSD material introduces UsdShade as the schema family used for creating and binding materials, where materials store renderer-facing shading definitions. The OpenUSD UsdShade specification defines UsdShadeNodeGraph as the mechanism for packaging shading networks into reusable units. A node graph can contain shader nodes and nested node graphs, expose public inputs as an interface, provide outputs, and be referenced into larger networks as a reusable building block.

Option C is correct because NodeGraph is explicitly designed for reusable and parameterized shading-network structure. It allows a pipeline to encapsulate repeated shading logic while exposing only selected parameters to consuming materials or larger graphs.

QUESTION 4
What key capability distinguishes an IsA schema from an API schema?

A. An IsA schema can impart a typeName to a prim, whereas an API schema cannot.
B. API schemas can have relationships, while IsA schemas cannot.
C. IsA schemas are always concrete, while API schemas are always non-concrete.

Answer: A
Explanation:
The key distinction is that an IsA schema, also called a typed schema, defines the fundamental type of a prim through its typeName metadata. NVIDIA's Learn OpenUSD glossary states that IsA schemas define a prim's fundamental type through typeName, and that each prim has only one IsA schema determining its core nature, such as Mesh, Camera, or Light. NVIDIA's schemas lesson also states that, unlike IsA schemas, API schemas do not assign a typeName; instead, they are list-edited in the apiSchemas metadata and queried using HasAPI. (NVIDIA Docs)

Option A is correct because it directly identifies the defining capability of an IsA schema: imparting a prim type. API schemas instead augment already-typed prims with additional properties or behavior.

QUESTION 5
Consider the following prim hierarchies. Each prim hierarchy is represented by nested unordered lists and each list item represents a prim with the following format: PRIM_NAME (KIND). No text within the parentheses means that the kind is unset for that prim. Only evaluate model kind hierarchy correctness and the structural choices. Which prim hierarchies represent valid model kind hierarchies? Choose three.

A. CarA (component)
   Interior ()
   SteeringWheel ()
   Chasis ()
   Wheels ()
   Wheel_FR ()
   Wheel_RL ()
B. CarA (assembly)
   Interior (group)
   SteeringWheel (component)
   Chasis ()
   Wheels (group)
   Wheel_FR (component)

Wheel_RL (component)

C. CarA (component)
Interior ()
SteeringWheel (component)
Chasis ()
Wheels ()
Wheel_FR (component)
Wheel_RL (component)

D. CarA (component)
Interior ()
SteeringWheel (subcomponent)
Chasis ()
Wheels (group)
Wheel_FR (subcomponent)
Wheel_RL (subcomponent)

E. CarA (component)
Interior ()
SteeringWheel (subcomponent)
Chasis ()
Wheels ()
Wheel_FR (subcomponent)
Wheel_RL (subcomponent)

Answer: ABE
Explanation:
The valid model kind hierarchies are A, B, and E. NVIDIA's Learn OpenUSD material defines group, assembly, and component as model kinds, while subcomponent is explicitly outside the model hierarchy. The hierarchy rule is that all ancestors of a component must be group or a subkind of group, such as assembly, and a component acts as a leaf or pruning point in the model hierarchy; therefore, it cannot contain descendant model kinds such as another component, group, or assembly.

A is valid because CarA is a component and every descendant has unset kind, so no nested model violates the component boundary. B is valid because CarA is an assembly, organizational containers use group, and the leaf model assets are component prims. E is valid because CarA is a component, and internal important parts are marked as subcomponent, which is permitted inside a component and does not count as a model kind.

QUESTION 6
Which of the following statements about debugging complex LIVRPS scenarios are correct?

A. Local opinions should be checked first when debugging composition issues.
B. Specialize arcs cannot be overridden, so they can be ignored during debugging.
C. Payload arcs should be checked before Reference arcs for more efficient debugging.

Answer: A
Explanation:
Option A is correct because local opinions are the strongest opinions in USD composition strength ordering. NVIDIA's Learn OpenUSD strength-ordering guide describes LIVRPS, now also written as LIVERPS with relocates included, as the ordered set of composition operations used to determine which opinions win. The order is strongest to weakest: local, inherits, variant sets, relocates, references, payloads, and specializes.

When debugging an unexpected value, local opinions should be inspected first because a directly

authored opinion in the local layer stack can override opinions arriving through other composition arcs. NVIDIA's tracing guidance also highlights how complex composition debugging becomes when many prims, properties, layers, and arcs contribute competing opinions.

QUESTION 7
Which of the following are immutable once a USD Stage has been opened? Choose two.

A. Rules for loading payloads from prims in the stage.
B. Layers that are muted or unmuted in the stage.
C. The path resolver context that is bound to the stage.
D. Variant fallbacks for prims without variant selection in the stage.

Answer: CD
Explanation:
The immutable stage-opening concerns are the path resolver context and the variant fallback configuration used for unselected variants. The resolver context is bound when the stage is created or opened and is used for future asset-path resolution on that stage, regardless of what other resolver context may be bound elsewhere. The OpenUSD UsdStage API documentation describes this binding behavior during stage creation, making option C correct.

Option D is also correct because global variant fallback preferences are defined as preferences "used in new UsdStages." Once a stage has been composed with its fallback preferences, changing global fallback settings affects newly opened stages, not the already-opened stage's established fallback behavior.

QUESTION 8
A key responsibility of the pipeline is to assemble content that may have divergent stage metadata across layer stacks. Which of these metadata is automatically handled by composition when they diverge across layer stacks?

A. timeCodesPerSecond
B. upAxis
C. kilogramsPerUnit
D. metersPerUnit

Answer: A
Explanation:
The metadata automatically handled by composition is timeCodesPerSecond. NVIDIA's Learn OpenUSD units guidance states the distinction directly: timeCodesPerSecond is automatically reconciled during composition, while metersPerUnit, kilogramsPerUnit, and upAxis are not. Pipelines must therefore explicitly handle geometric orientation, spatial scale, and physics mass-unit mismatches when assembling assets from different sources.

Option A is correct because OpenUSD can automatically scale time-sampled values across sublayer, reference, and payload arcs when source and target layer stacks use different timeCodesPerSecond values. OpenUSD's time-value documentation explains that when a targeted layer specifies timeCodesPerSecond, TimeCode coordinates and animated value coordinates are automatically scaled into the time frame defined by the source layer's timeCodesPerSecond.

QUESTION 9
Suppose you had the following layer:

```
#usda 1.0

(

defaultPrim = "ParentXform"

)

def Xform "ParentXform"

{

def Mesh "ChildMesh"

{

}

}
```

If you wanted to add a property to "ParentXform" such that it would automatically propagate to "ChildMesh" without having to add the same property to "ChildMesh", which of the following changes to "ParentXform" would make this work?

A. Add the property as a custom attribute:
custom string myProperty = "TestValue"
B. Add the property as a relationship to </ParentXform/ChildMesh>:
rel myProperty = </ParentXform/ChildMesh>
C. Add the property as a primvar, with "constant" interpolation:
string primvars:myProperty = "TestValue" (
interpolation = "constant"
)

Answer: C
Explanation:
The correct mechanism is a constant-interpolation primvar. NVIDIA's Learn OpenUSD glossary defines primvars as primitive variables authored using the primvars: namespace and interpolation metadata such as constant, uniform, vertex, and faceVarying. Primvars are designed to carry geometric or shading-related data in a way that consumers can discover through UsdGeomPrimvarsAPI and UsdGeomPrimvar.

Option C is correct because constant primvars can inherit down the namespace hierarchy. The OpenUSD UsdGeomPrimvar documentation states that constant interpolation primvar values can be inherited by child prims unless those children author their own opinion for the same primvar. This allows primvars:myProperty authored on /ParentXform to be discovered on /ParentXform/ChildMesh without duplicating the property on the mesh.

QUESTION 10
You are setting up an outdoor scene with realistic lighting and want to simulate the effect of the sun. Which UsdLux light type is most appropriate for this purpose, providing a directional light source with parallel rays?

A. RectLight

B. DomeLight
C. DistantLight
D. SphereLight

Answer: C
Explanation:
The correct UsdLux light type is DistantLight. NVIDIA's Learn OpenUSD lighting guide identifies UsdLux.DistantLight as a schema where light is emitted from a distant source along the negative Z axis, commonly known as a directional light. This makes it the most appropriate light type for representing sunlight in an outdoor scene.

Option C is correct because sunlight is effectively modeled as coming from an extremely distant source, so its rays are treated as broadly parallel across the scene. The OpenUSD DistantLight schema documentation explicitly describes it as typically used for distant, broad light sources like sunlight, where rays affect the entire scene and are roughly parallel. It also notes that the inputs:angle attribute controls angular diameter; the fallback value approximates the Sun's apparent angular size from Earth.

QUESTION 11
What will be the composed value of /World/Tree/Canopy.primvars:displayColor when you open stage.usda?

```
#usda 1.0

(

defaultPrim = "World"

metersPerUnit = 1.0

upAxis = "Z"

)

def Xform "World"

{

def Xform "Tree" (

variantSets = ["foliage_color"]

variants = { string foliage_color = "default" }

)

{

def Cone "Canopy" (

references = [</_base_foliage_color>]

)

{
```

```
double3 xformOp:translate = (0, 0, 1.3)
token[] xformOpOrder = ["xformOp:translate"]
}
def Cylinder "Trunk"
{
color3f[] primvars:displayColor = [(0.2, 0.1, 0.05)]
double3 xformOp:scale = (0.4, 0.4, 0.4)
token[] xformOpOrder = ["xformOp:scale"]
}
variantSet "foliage_color" = {
"default" {
}
"evergreen" {
over "Canopy"
{
color3f[] primvars:displayColor = [(0.05, 0.15, 0.05)]
}
}
"orange" {
over "Canopy"
{
color3f[] primvars:displayColor = [(0.5, 0.3, 0.05)]
}
}
}
}
}
class "_base_foliage_color"
```

```
{

color3f[] primvars:displayColor = [(0.2, 0.75, 0.1)]

}
```

A. [(0.5, 0.3, 0.05)]
B. [(0.2, 0.75, 0.1)]
C. unset
D. [(0.05, 0.15, 0.05)]

Answer: B
Explanation:
The composed value is [(0.2, 0.75, 0.1)]. The active variant selection on /World/Tree is foliage_color = "default", and the "default" variant contains no authored opinion for Canopy.primvars:displayColor. Therefore, the "evergreen" and "orange" variant opinions are not active and cannot contribute their color values. NVIDIA's Learn OpenUSD variant-set guidance explains that a variant set provides alternative authored representations, but only the selected variant contributes to the composed result.

The remaining contributing opinion is the reference on def Cone "Canopy": references = [</_base_foliage_color>]. Reference are composition arcs that bring scene description from a targeted prim into the destination prim without copying it. The referenced class prim /_base_foliage_color authors color3f[] primvars:displayColor = [(0.2, 0.75, 0.1)], so that value composes onto /World/Tree/Canopy. No stronger local or selected-variant opinion overrides it. This follows USD strength ordering, where opinions are resolved through composition arcs and the strongest applicable opinion wins. This aligns with Composition -> Reference, Variant Sets, Selected Variants, and LIVERPS Strength Ordering.

QUESTION 12
Which of the following are true for SdfChangeBlocks? Choose two.

A. Notifications are muted for changes done within the block, even after the block exits
B. Existing metadata and properties can be changed using USD APIs within the same block
C. It is unsafe to query Prims that are mutating within the same block
D. It is unsafe to delete or create new PrimSpec hierarchies using Sdf APIs

Answer: C
Explanation:
Strictly evaluated against the OpenUSD API contract, C is the only fully correct statement as written. SdfChangeBlock groups multiple low-level Sdf edits so that change processing can be delayed and handled efficiently as one batch. The official API documentation states that opening a change block causes Sdf to delay notifications until the outermost block exits; notifications are queued, not permanently muted. Therefore, A is incorrect. The same documentation gives a strong warning that it is not safe to use Usd or other downstream APIs while a change block is open, because those representations may be stale or may hold expired handles to Sdf objects. Therefore, querying mutating prims during the block is unsafe, making C correct.

Thank You for Trying Our Product

Lead2pass Certification Exam Features:

★ More than **99,900** Satisfied Customers Worldwide.

★ Average **99.9%** Success Rate.

★ **Free Update** to match latest and real exam scenarios.

★ **Instant Download** Access! No Setup required.

★ Questions & Answers are downloadable in **PDF** format and **VCE** test engine format.

★ Multi-Platform capabilities - **Windows, Laptop, Mac, Android, iPhone, iPod, iPad**.

★ **100%** Guaranteed Success or **100%** Money Back Guarantee.

★ **Fast**, helpful support **24x7**.

View list of all certification exams: http://www.lead2pass.com/all-products.html

10% Discount Coupon Code: **ASTR14**