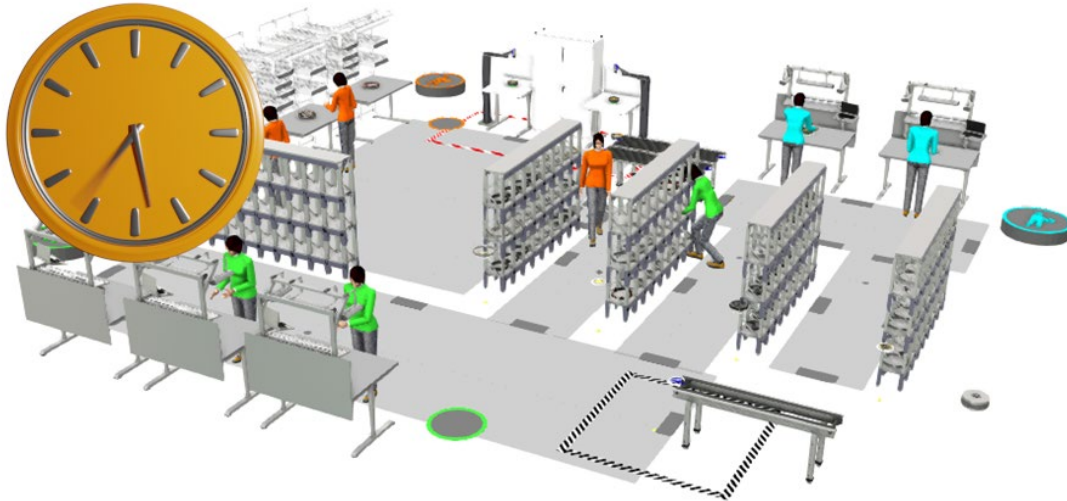


Optimizing Simulation Performance

Visual Components 4.6 | Version: February 15, 2023



Simulating larger layouts with increased speed, requires consideration when creating the components and layout. This lesson will introduce the basic concepts used to achieve the best possible simulation performance.

In this tutorial, you will learn:

- Concepts of geometry simplification
- Concepts of implementing simulation behavior for better performance
- Running simulations with proper settings
- Taking performance into account when developing custom component scripts
- How to profile the performance of a layout

Support

support@visualcomponents.com

Visual Components Forum

forum.visualcomponents.com

Contents

Geometry.....	3
Geometry Importing	3
Geometry Simplification.....	4
Addons for geometry cleanup.....	6
Geometry Memory Consumption	6
Conclusion regarding geometry	7
Simulation Behavior	7
Simplify the simulation behavior.....	7
Prefer component behaviors over Python scripting.....	8
Model rebuilding and updating.....	8
Some best practices for model design	9
Running the simulation	10
Simulation clock setting	10
Rendering mode.....	10
Joint limit checking.....	11
Statistics interval	11
Simulation Level	12
Python scripts.....	13
Utilize events in scripts.....	13
Store objects to variables.....	13
Using Print statement	14
Rebuilding	14
Hiding and showing geometries during simulation	15
Updating the scene	15
Creating dynamic components.....	15
Reuse objects	16
Avoid context switch	16
Profiling the layout performance.....	16
Profiler Addon	16

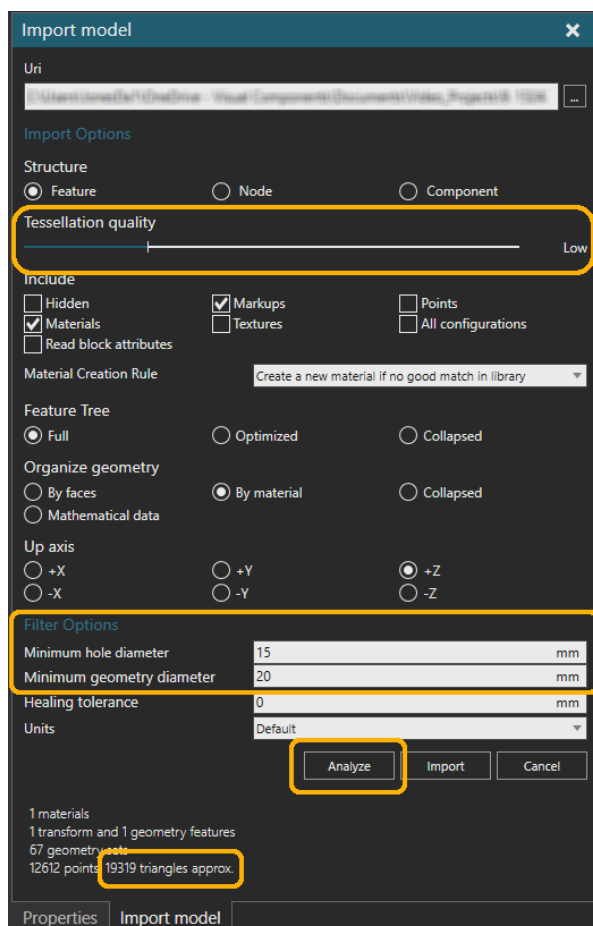
Geometry

Although rendering is not always a bottleneck, it is recommended to create models that are as lightweight as possible. When creating large layouts, it is better to exclude all details from each component in the scene. This includes all nuts and bolts and other tiny fixtures that do not add value to the simulation. Often CAD models from machine vendors or mechanical design departments are too detailed, and so must be simplified.

Geometry Importing

Make the right choices at the beginning. If possible, try to obtain an already simplified CAD model. Use as low a **Tessellation quality** as possible when importing a CAD model. And it is important to use the options included in **Import model** panel, to filter out the small geometry items and holes already at the import phase.

Before importing a model, use the **Analyze** command in the **Import model** panel to check how much data you are about to import. Pay extra care to any models you plan to clone multiple times in the layout, like for examples CNC machines, conveyors, fences, etc.



Feature Tree defines what hierarchy to use for geometry.

- If **Full** is selected, an attempt will be made to match the structure of the model when viewed in its native CAD editor. **Full** creates feature tree based on CAD's assembly tree, meaning that each item in the assembly tree will be its own feature. A feature containing geometry generates a geometry feature, otherwise a transform feature.

- **Optimized** generates a flat structure of feature tree, without extra transform features.
- **Collapsed** creates only one geometry feature. This option can be used to reach optimal performance when detecting the geometry with collision check, ray cast and volume detection.

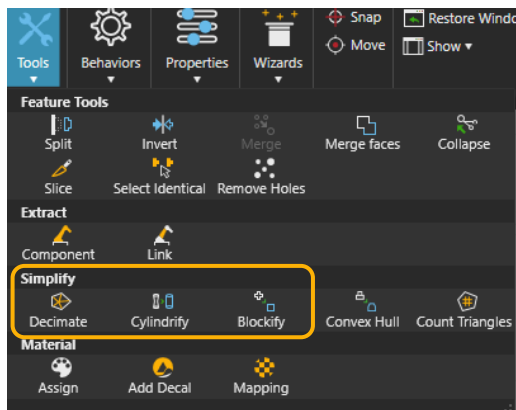
Organize geometry defines how the geometry will be organized into geometry sets. Geometry sets can be sliced, exploded, or collapsed.

NOTE: Use **By material** or **Collapsed** as default options.

- **By faces** will make each face as own geometry set. This is not recommended unless needed for specific topology use cases, so only use if required.
- **By material** option creates own geometry set for each color in a single mesh. Some CAD formats can define different colors in different faces in a single mesh, but in Visual Components there can be only one material per geometry set.
- **Collapsed** combines all the meshes into one geometry set. However, to optimize the performance, a new set will be created when the total amount of triangles exceeds 10,000 (or 16,000 points).
- **Mathematical data** will organize geometry one set per face and store a BREP entity in a triangle set creating a larger geometry and file size. Used for special scenarios, for example when special accuracy is needed in robot offline programming.

Geometry Simplification

Once a model has been imported, it can be simplified further. Consider using the **Cylindrify** and **Blockify** commands, available from the **Tools** option in the **MODELING** tab.

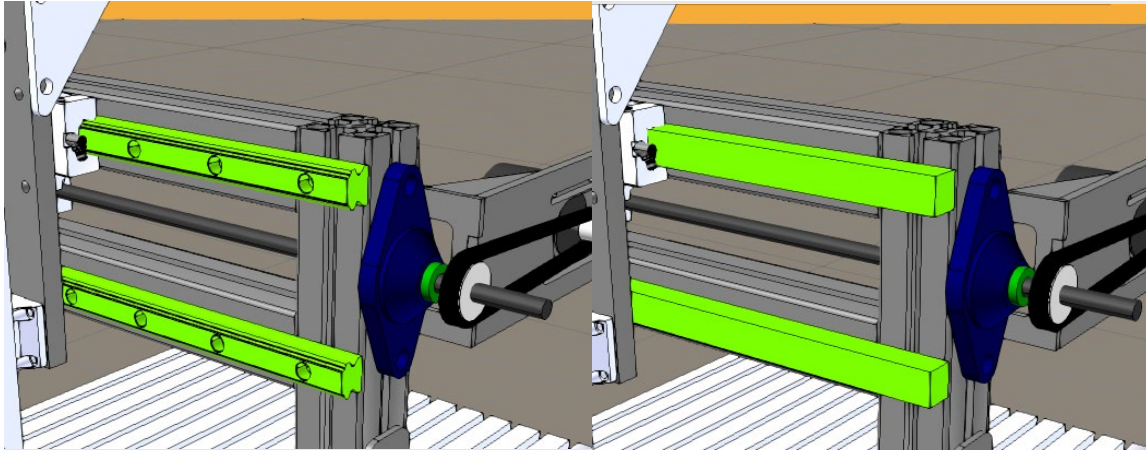


Those commands will convert the selected geometry sets to simple cylindrical or block shapes, and in some cases saving thousands of triangles without losing too much of the overall shape of the device.

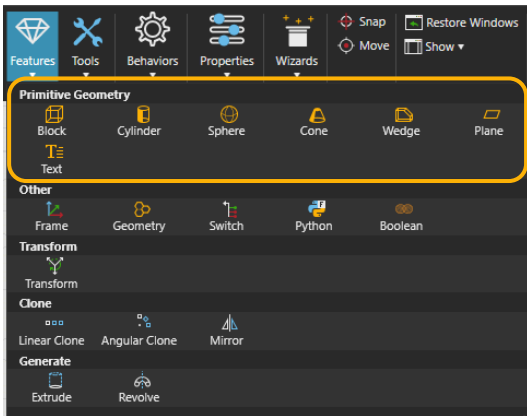
NOTE: Be careful with these commands as they are destructive and cannot be undone. Create backup copies of layouts including imported models, before you begin using these options.

Also consider using **Decimate** command, that attempts to create the same shape as the selected geometry sets but with less triangles. And the same applies to **Decimate** as for the **Cylindrify** and **Blockify** commands. The undo action cannot be used with these.

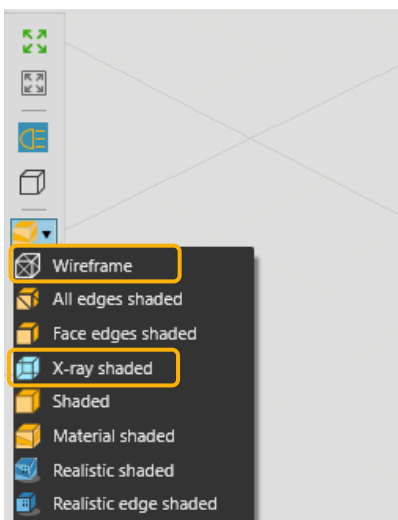
The original and the result of using the **Blockify** command is shown in the image below. The triangle count for each guide rail decreased from 2276 triangles down to 24 triangles.



Also consider if some shapes can be represented with simple representation, by using the primitive geometries available inside the Visual Components (VC) application.



Also remove any details that nobody will ever see. Like details inside electric motors or behind covers that are never revealed during the simulation. The **Wireframe** or **X-ray shaded** rendering modes are great for checking if something unnecessary is hiding inside the imported CAD model.

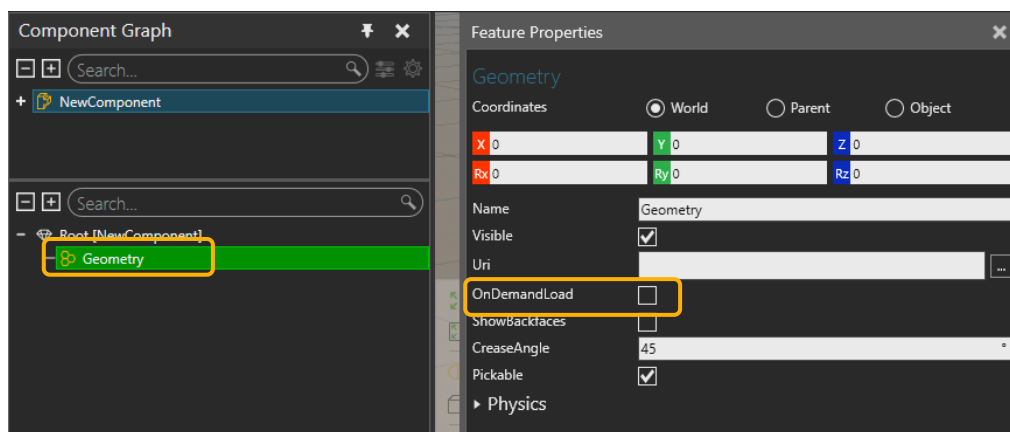


Addons for geometry cleanup

On the [Visual Components Forum](#), you can find addons that help the process of geometry simplification. *Triangle Count* addon helps to find the most geometry heavy components and features in the layout. *Auto Materialize* addon helps to remove unnecessary tiny details or to decimate shapes with certain sizes.

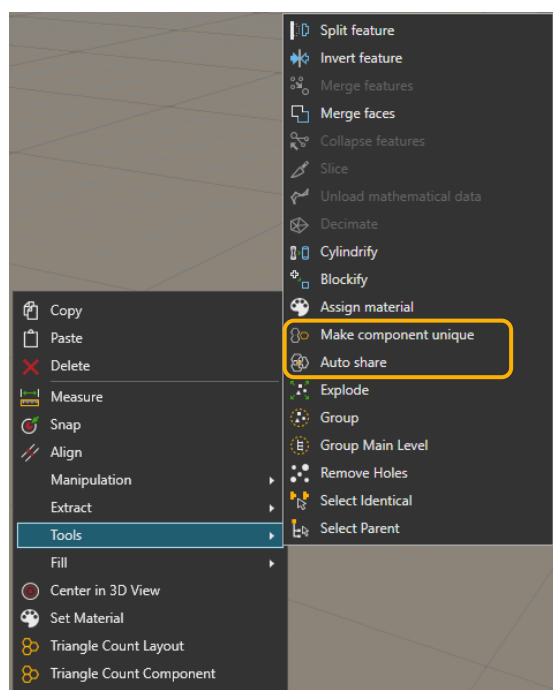
Geometry Memory Consumption

The Geometry feature inside the VC application has a property called **OnDemandLoad**. When enabled, the respective geometry is loaded into memory only when the geometry must be visualized. So for example, if the geometry is hidden with the help of switch feature, the geometry is not loaded into the scene until it is needed. This will make loading and saving models faster.



When cloning (or copy/paste) components in the scene, the geometry of the cloned instances is shared with the original. **Shared geometry** is easy to notice on the modelling tab when selecting a feature, and the same selected feature is highlighted (green) in all cloned instances. This is intentional and helps to consume less memory in the scene.

Auto Share and **Make component unique** commands can be found in the 3D context menu under Tools sub menu.



If the sharing must be broken, to for example edit the geometry of one cloned instance, the **Make Component Unique** command can be used. Also changing any property that forces a model to rebuild (e.g., ConveyorLength or component Material) will also break the sharing. If the cloning is broken, but there are similar instances of the same component, the **Auto Share** command can be used to reshare the geometry representation of similar components in the scene.

Auto Share is a great tool to share the geometries of similar components in the scene that were not originally cloned. If e.g. two similar conveyors are loaded to the scene from the eCatalog which have the same property settings in the scene, they will share their geometry representation if the **Auto Share** command is used. Shared geometries will also improve the rendering performance.

Conclusion regarding geometry

In general, heavy geometry will affect the overall performance of the simulation. Affecting these (but not limited to these) aspects:

- Rendering performance.
- Performance of the collision check, ray casting and volume detection.
- Performance of the possible geometry rebuilds.
- Layout loading and saving times.
- Also, heavy geometry consumes memory, which might be a limiting factor in some cases.

Simulation Behavior

The way simulation behavior is implemented, is the most critical factor affecting overall simulation performance. Creating the model in the *correct* way and simulating only the necessary details is the way to achieve the best possible simulation performance.

Simplify the simulation behavior

Simulating only that which adds value is a great starting point:

- Before creating the model, think again about why it was created.
- If the purpose is not to create videos with close ups of details, consider excluding *all possible* unnecessary details.
- Do not add details that do not serve a purpose. Exclude simulation (animating, adding motion) of human walk cycles, parts dropping into bins with gravitation, complex clamp mechanisms and small details inside the machine that nobody will ever see.
- Avoid creating components that can do *everything*. A versatile component that can be used in all use cases is great, but it is also extremely hard to model such logic without sacrificing simulation performance.

The components in the public online library (**eCatalog**) provided by Visual Components are modeled for a generic purpose. They are modeled to be suitable for as many use cases as possible, and this versatility sacrifices the simulation performance a little. Although in

general, the models in the online library can be considered to be modeled the “correct” way. In some cases, consider creating a simple single purpose component, instead of utilizing one of the flexible online library components.

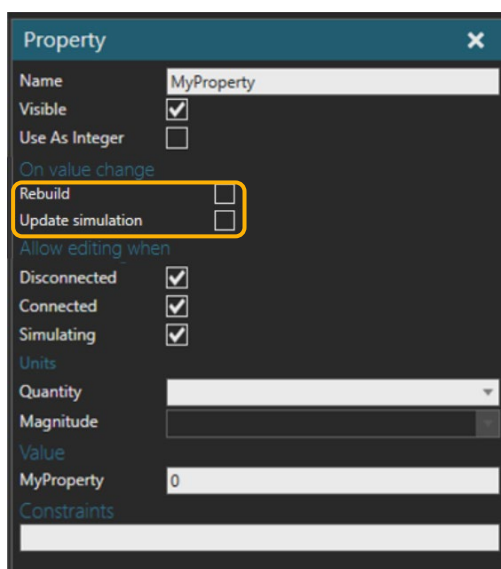
After all, we are creating visual simulations in Visual Components, so we do not exclude all the visual aspects from the simulation, so that the simulation will communicate the intended results in an understandable way. Just start creating the model by adding the basic functionality and the overall flow. Then at the end, add all the necessary details.

Prefer component behaviors over Python scripting

Custom Python scripts allow you to extend the functionalities of models when no built-in behavior is available. However, *whenever possible, use the built-in behaviors*, since they are executed in the simulation core, and thus do not require a performance-heavy 'context switch' (i.e., Visual Components needs to switch between the execution of the simulation core and the Python engine). Therefore, even the most optimized Python script is slower than the built-in behaviors.

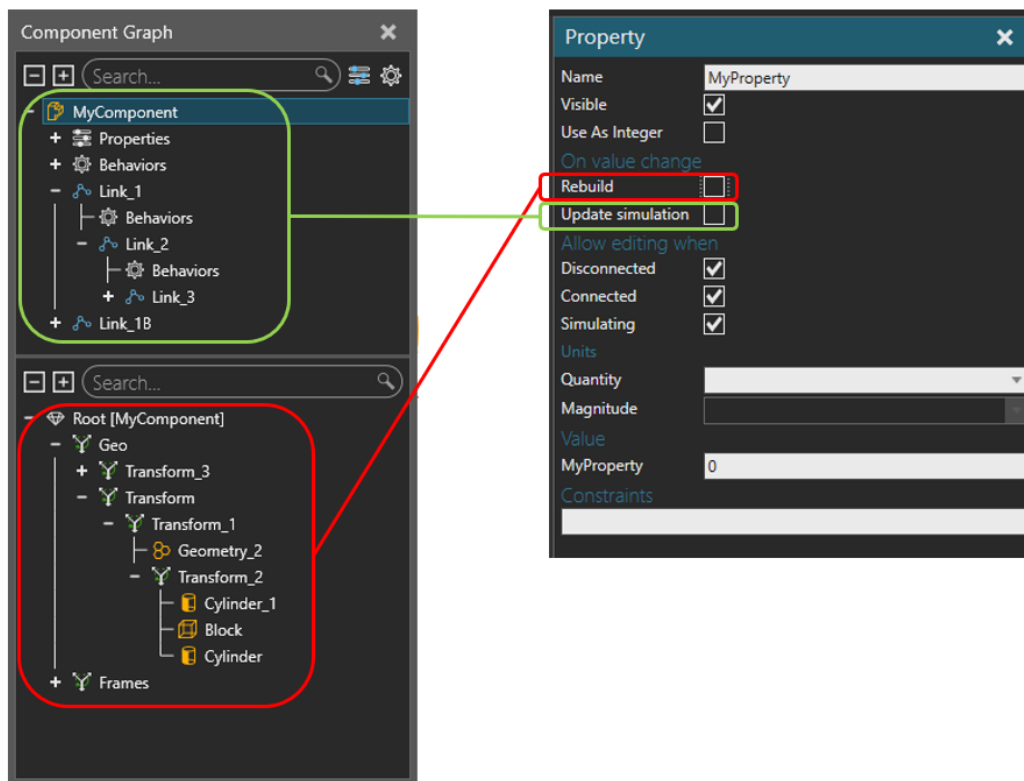
Model rebuilding and updating

When creating new component properties, disable **Rebuild** and **Update simulation** options if they are not needed. If **Rebuild** is checked, the model geometry tree is re-evaluated and re-generated when the property value is changed. This is necessary for properties like conveyor height, to see the value change effect in the component geometry.



However, if there is a property that is used for instance as a counter during simulation, leaving a **Rebuild** flag on for such a property can seriously affect performance, as the geometry of the component is re-evaluated each time the counter value is changed during the simulation. Design the components so that they will not need rebuilding during simulation. Implementing dynamic geometrical changes (like moving links) must be implemented using the node (i.e., link) tree in the component graph instead of the feature tree. Update Simulation should be disabled also if the property is not needed in the link tree. However, simulation update for one single component is fast and does not effect on the simulation performance as much as rebuilding.

For example, if a frame feature must be repositioned during the simulation, it is better to place the frame under a link that is moved during the simulation instead of placing the frame feature under a transform feature which is modified with a parametric expression. Updating the transform feature requires a rebuild but updating the location of the link requires only the simulation update. See the image below how property settings relate to the expressions in the component structure in the component graph.



There is an addon on the [Visual Components Forum](#), that analyzes all the components in the scene and disables the **Rebuild** flags of the properties that are not used in the feature tree expressions. This helps prevent unintentional rebuilds but does not fix badly designed components that require rebuilds during simulation run. Search for addon names “[Optimize the property Rebuilds](#)”

Some best practices for model design

- *Create all necessary properties, features and behaviors in design phase.* Do not create them during runtime dynamically in scripts. This applies to the properties in the dynamic components (i.e., products on the production line) too. Create the necessary properties to the product components before running the simulation if possible.
- *Fix all expression issues.* Geometry feature tree or Component link tree may have complex expressions. This is not considered as a bottleneck to the performance. However, if there are errors in the expressions it is a major performance issue. Errors are printed on the output panel. Each time an error occurs, the expression is re-evaluated, and this causes performance issues.
- Prefer using *built-in behaviours over custom scripts* when possible. Use signals, sensors, paths, capacity controllers, routing rules etc.

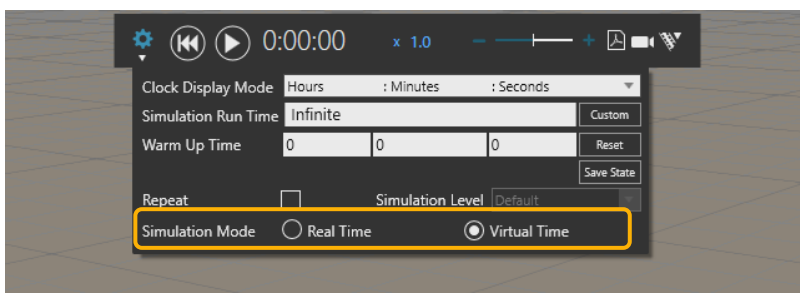
- Use *Process Modeling feature*. Much of the simulation flow and logic of machines and devices can be modeled with the Process Modeling capability. Familiarize yourself with the Process Modeling feature with the lessons and courses available on the Visual Components Academy, and try to implement most of the simulation utilizing those capabilities and minimize the amount of custom scripting.
- In Robots the *Python kinematics solver is slower than built-in kinematic solvers*. If possible use the built-in solvers, and in projects where you can more freely choose a robot model, try to choose one that does not use Python kinematics. Also, if possible, try to use more PTP (Point to Point) type motions instead of linear motions with robots with inverse kinematics, as it is a less calculation demanding motion type.
- Avoid using *Raycast and Volume Sensors if possible*, instead prefer e.g. Path Sensors. If Raycast or Volume Sensor types are required, pay attention to the frequency those sensors are utilizing by adjusting the sample time property, or consider disabling sampling and connect a signal to the sensor to trigger the sensor only when it is needed.
- In conveyor systems, use Accumulate, RetainOffset and SpaceUtilization in the path behaviors, only if they are really needed

Running the simulation

Although the most crucial factor to simulation performance is the way the model is built, there are still certain things that must be considered when running simulations to gain maximum performance.

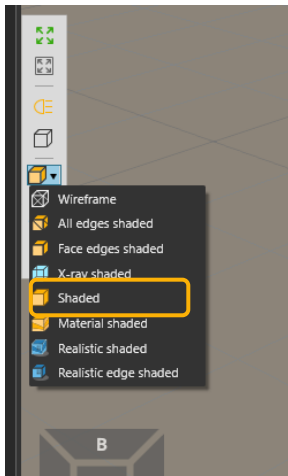
Simulation clock setting

Simulation run settings can be found on the top of the 3D panel. In the settings, set the **Simulation Mode** to **Virtual Time**. It is slightly faster to run the simulation in the **Virtual Time** mode than **Real Time**, as the simulation does not need to match the simulation time to real (Windows) clock.



Rendering mode

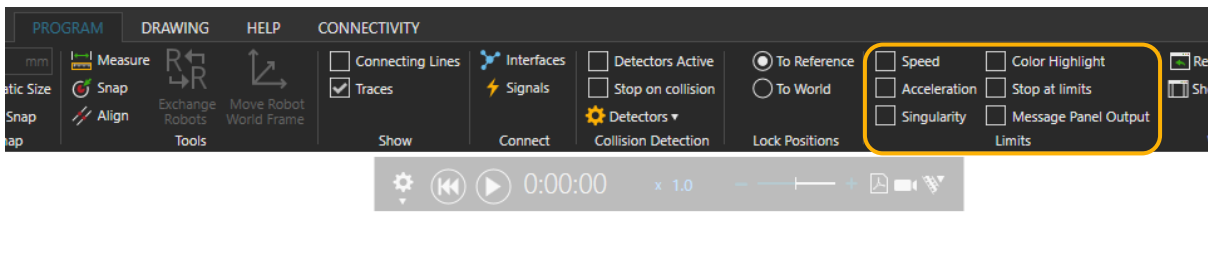
When simulating fast, the rendering is not called that often and the scene is rendered only to keep the 3D window up to date. But even so using the simpler **Shaded** rendering mode, compared to more demanding rendering modes like **Realistic Shaded**, will increase the simulation performance. This is a more noticeable improvement in scenes with a large amount of 3D data.



Joint limit checking

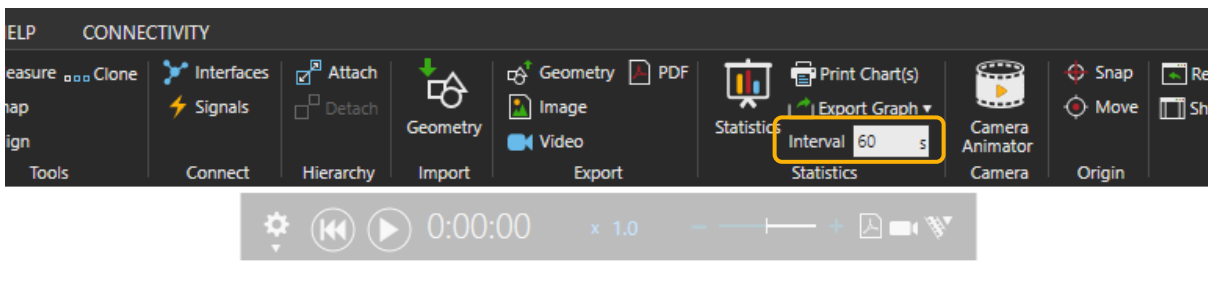
Making sure that none of the joints in the devices are exceeding their limit values is often one reason to simulate. Joint limit checking is however a bit of a demanding task for the simulation engine. If the joint limits are already being checked and verified, it is recommended to disable the limit checking to run simulations faster.

Disable joint limit checking when it is not needed. Disable all joint limit options on the Program tab. This one setting applies not only to all robots in the scene, but to all devices that have servo joints and motions.



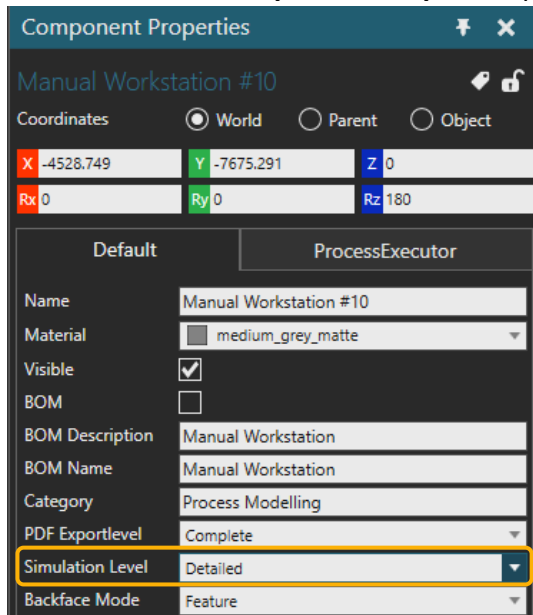
Statistics interval

Gathering statistics is essential when simulating to analyse the simulation either after or during the simulation. However, gathering statistics with a short interval will impact the simulation performance. If the simulation run is long, like multiple weeks, the statistics data may not be needed with the default 60s interval. *Consider increasing the statistics Interval in the Home tab ribbon bar.*

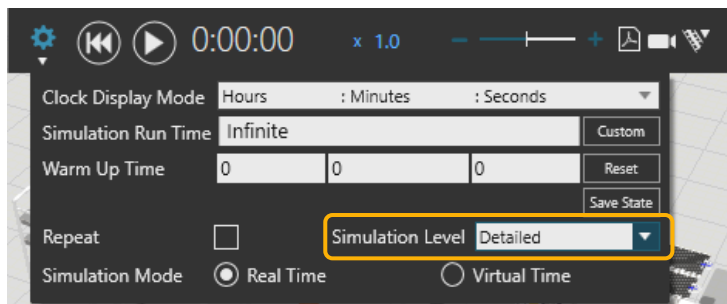


Simulation Level

Use the **Simulation Level** feature in multipurpose components. **Simulation Level** option can be found in the **Component Properties** panel for each component.



And a **Simulation Level** option is available under the simulation configuration settings for the whole layout level (applies to all components). Simulation behavior for each component must be implemented separately to support **Simulation Level** feature.



For example, in **Fast** mode some servo motions can be replaced with simple delays in scripts. The only behavior that automatically uses **Fast** mode is a physics cable. The rest of the support is in Python scripts. Some online catalog library models utilize this feature by controlling the level of simulation details in the component implementation. For example, the Process Modeling resources in the online catalog utilize the simulation level feature. If there are no components in the scene that utilize the **Simulation Level** feature, the option in the settings is disabled.

Add this event method to a component script to register the component to utilize the **Simulation Level**

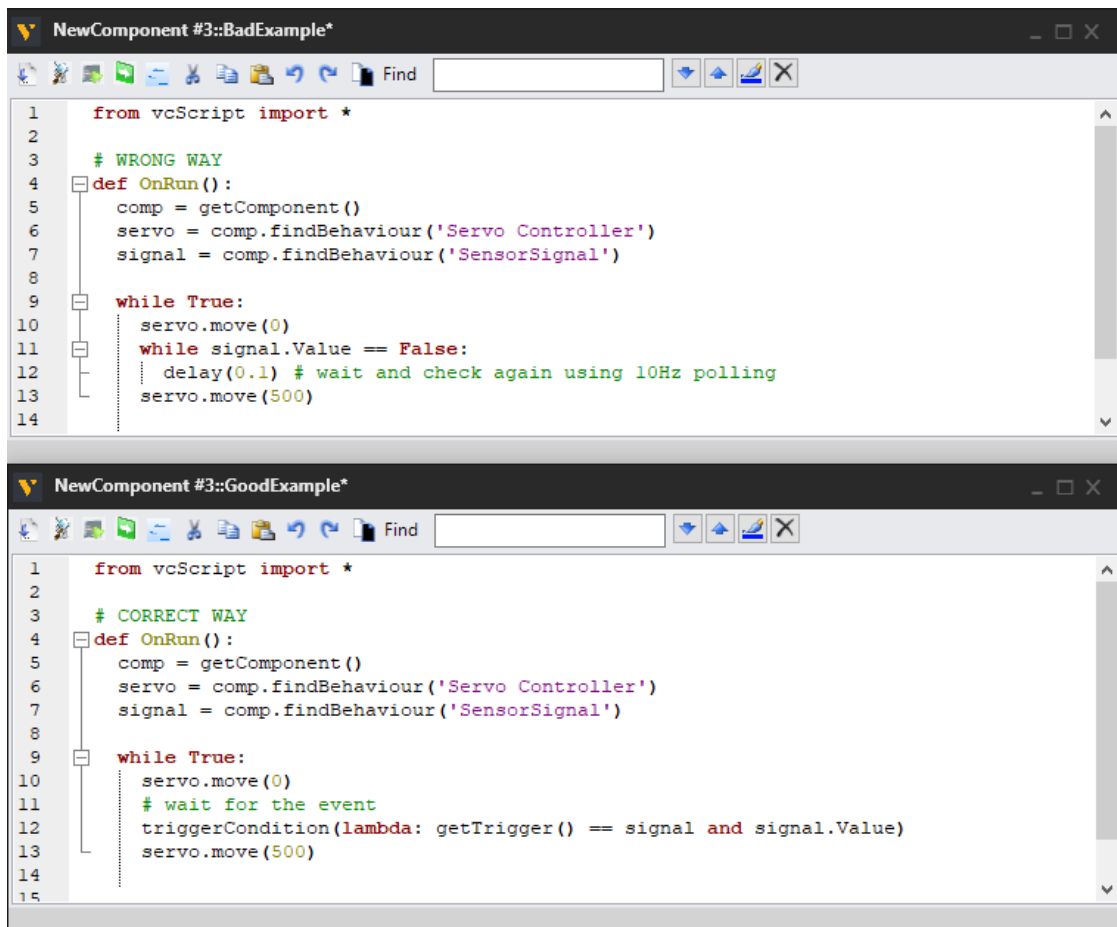
```
def OnSimulationLevelChanged(level):  
    pass
```

Python scripts

In general, it is better to implement simulation logic with built-in behaviors, statements in the Process Executors and/or in Robot programs. But to achieve the necessary logic Python scripts are often needed. To maximize the simulation performance certain aspects in scripts must be considered.

Utilize events in scripts

Instead of testing certain conditions repeatedly (polling) with a high frequency, it is better to utilize events and model the logic with discrete events. In the example scripts below, above is an example of a polling logic not preferred, while below the same logic is implemented with an event-based approach, and is faster to execute and will give a more accurate result.



The image shows two screenshots of a Python script editor. The top screenshot, titled 'NewComponent #3::BadExample*', shows a script with the following code:

```
1 from vcScript import *
2
3 # WRONG WAY
4 def OnRun():
5     comp = GetComponent()
6     servo = comp.findBehaviour('Servo Controller')
7     signal = comp.findBehaviour('SensorSignal')
8
9     while True:
10        servo.move(0)
11        while signal.Value == False:
12            delay(0.1) # wait and check again using 10Hz polling
13        servo.move(500)
14
```

The bottom screenshot, titled 'NewComponent #3::GoodExample*', shows a script with the following code:

```
1 from vcScript import *
2
3 # CORRECT WAY
4 def OnRun():
5     comp = GetComponent()
6     servo = comp.findBehaviour('Servo Controller')
7     signal = comp.findBehaviour('SensorSignal')
8
9     while True:
10        servo.move(0)
11        # wait for the event
12        triggerCondition(lambda: getTrigger() == signal and signal.Value)
13        servo.move(500)
14
15
```

Store objects to variables

Instead of using `GetComponent()`, `getSimulation()`, `getApplication()`, `findBehavior()`, `getProperty()` methods all over the script and calling these methods during run time, cache the objects to variables. Calling these methods to get the handle to the object during simulation will slow down the simulation performance. It is better to obtain all objects and store them into variables before the actual simulation logic starts. Usually, there is a while loop inside the `OnRun` function. Obtain the required objects before the while loop like shown in the image below.

```
NewComponent #3::CacheObjects
1  from vcScript import *
2
3  app = getApplication()
4  sim = getSimulation()
5  comp = getComponent()
6
7  def OnRun():
8      path = comp.findBehaviour()
9      controller = app.findComponent('LayoutController')
10     task_signal = controller.findBehaviour('Task')
11     interval_prop = comp.getProperty('Interval')
12     frame_feature = comp.findFeature('ResourceFrame')
13     location = frame_feature.NodePositionMatrix
14
15     while True:|
16         ...
17         ...
18         ...
19         ...
20
```

Using Print statement

Although, printing messages into the **Output** panel when developing Python scripts is useful, it is recommended to remove all unnecessary printouts from completed scripts, as they will influence the performance of the simulation. On top of this, a clean **Output** panel is easier to follow than a panel full of test prints from different components in the layout.

Rebuilding

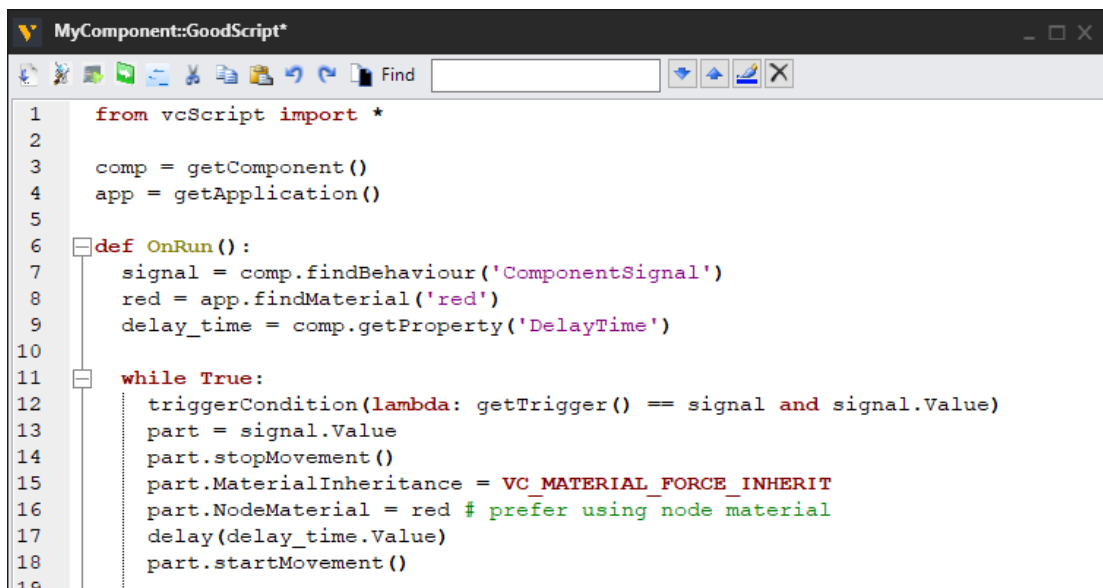
As mentioned in the [Simulation Behavior](#) section of this lesson, the rebuilding of geometry is heavy, so this must also be considered when developing scripts. Create the logic and the structure of the models so that the `vcComponent.rebuild()` or `vcFeature.rebuild()` methods are not needed during the simulation run.

Also, when switching the color (i.e., material) of a component during simulation, avoid using the `vcComponent.Material` property. Use `vcNode.NodeMaterial` instead, as the `NodeMaterial` changing does not require component rebuild. Refer to the bad and good example below.

In the *bad example* the necessary objects are obtained during the simulation run in the while loop repeatedly. Also, the component material of the part object is changed during simulation that forces the component object to rebuild.

```
MyComponent::BadScript
1  from vcScript import *
2
3  def OnRun():
4      while True:
5          signal = getComponent().findBehaviour('ComponentSignal')
6          triggerCondition(lambda: getTrigger() == signal and signal.Value)
7          part = signal.Value
8          part.stopMovement()
9          red = getApplication().findMaterial('red')
10         part.Material = red # avoid using component material
11         stopdelay = getComponent().getProperty('DelayTime').Value
12         delay(stopdelay)
13         part.startMovement()
```

In this *good example* the objects are stored into variables before the while loop and instead of `vcComponent.Material` property, the `vcNode.NodeMaterial` property is changed. This won't require rebuilding, but requires setting of the material inheritance to define where the node material is applied in the model. `Force inherit` overwrites all material settings in the part object. This *good example* is also more readable and easier to maintain.



```
1  from vcScript import *
2
3  comp = getComponent()
4  app = getApplication()
5
6  def OnRun():
7      signal = comp.findBehaviour('ComponentSignal')
8      red = app.findMaterial('red')
9      delay_time = comp.getProperty('DelayTime')
10
11     while True:
12         triggerCondition(lambda: getTrigger() == signal and signal.Value)
13         part = signal.Value
14         part.stopMovement()
15         part.MaterialInheritance = VC_MATERIAL_FORCE_INHERIT
16         part.NodeMaterial = red # prefer using node material
17         delay(delay_time.Value)
18         part.startMovement()
19
```

Hiding and showing geometries during simulation

If certain parts of the model geometry must be hidden or shown during the simulation, it is better to use the component link tree again instead of the feature tree. Avoid using the switch feature for showing and hiding items during simulation. Switch feature is great for parametrization of component that does not need to change during simulation.

For example, to model a conveyor that can be configured to represent a roll conveyor or a belt conveyor. To show and hide geometry during simulation run, add links and parts of the geometry that must be shown or hidden under those links. Then during simulation use `vcNode.Visible` property to show and hide the items. This is extremely fast compared to switch feature that requires geometry rebuild.

Updating the scene

Some modifications in the scripts require updating. For example, to relocate a node using `vcNode.PositionMatrix` property requires a node update after setting a new matrix value. Instead of using `vcSimulation.update()`, use `vcNode.update()` to update only that node that requires updating.

Creating dynamic components

As mentioned in the [simulation Behavior section](#) of this lesson, using the built-in functionality of the built-in behaviors is faster than replicating the same in a script. When dynamic components must be generated for the simulation, it is better to use Component Creator and Product creator behaviors rather than calling `vcApplication.cloneComponent()` method during the simulation.

Also, try to add all properties, features and behaviors to the dynamic components before generating them into the line. Use `vcComponentCreator.TemplateComponent` to access the

component to be created before creating it. Or when using vcProductCreator use Product Editor on the **PROCESS** tab to edit the products to be created.

Keep dynamic components as simple as possible as there can be thousands of instances of the same component on the line. Also, avoid adding Python scripts and other behaviors to the dynamic components.

Reuse objects

Reuse objects as much as possible instead of recreating them. For example, vcMatrix objects can be reused, instead of re-creating them repeatedly with vcMatrix.new() constructor. Use vcMatrix.identity() method to reset an existing matrix to default *zero location*.

Similarly, vcMotionTarget and vcAction objects can be recycled to improve performance.

Avoid context switch

When simulation execution must switch between the simulation core and Python execution, it impacts the simulation performance negatively. Context switch from the Python to simulation engine happens every time the simulation must update or consume simulation time. For instance, when using delay() method, the execution from Python script must jump to simulation core to consume the given simulation time, and after that jump back to Python execution.

It is better to do as much Python calculation as possible at the same time, rather than a lot of context switches. Often context switches are hard to avoid, but it is good to understand the concept when trying to optimize the last bits of the simulation performance.

```
# two context switches
...
delay(5)
my_variable = 100.0
delay(5)
...

# one context switch
...
my_variable = 100.0
delay(10)
...
```

Other examples of methods forcing context switch are: vcServoController.move(), triggerCondition(), condition(), delay(). In addition, vcSimulation.update() causes context switch in certain layout/component configurations.

Profiling the layout performance

When the layout is done and the performance should be improved, but it may be hard to know where to start the profiling.

Profiler Addon

There is a profiling addon available on the forum that will reveal which components in the layout are most demanding, and it helps in understanding where the bottle neck in the simulation performance may be.

The “[Profiler](#)” addon will give a result that shows the total system time spent and the top 20 most performance expensive components in the layout, along with the number of calls to component behaviors including Python scripts and the amount of system time spent per component. All data (beyond the top 20 list) is dumped to a file located in a folder shown in the output message.